Blox

Final Report

December 20, 2016
Programing Languages and Translators
Professor Stephen Edwards

Group Members

Name	UNI
Naeem Bhatti	bnb2115
Jonathan Voss	jcv2130
Tyrone Wilkinson	trw2119

Contents

1 Introduction

2 Language Reference Manual
- 2.1 Lexical Conventions
- 2.2 Operators
- 2.3 Data Types
- 2.4 Expressions
- 2.5 Scope
- 2.6 Statements
- 2.7 Functions

3 Architecture
- 3.1 Source Code
- 3.2 Ast
- 3.3 Scanner
- 3.4 Parser
- 3.5 Analyzer
- 3.6 Executor
- 3.7 Generator
- 3.8 Gen Code

4 Project Plan
- 4.1 Group-Member Responsibilities
- 4.2 Project Goals
- 4.3 Challenges Faced

5 Lessons Learned
- 5.1 Naeem Bhatti
- 5.2 Jonathan Voss
- 5.3 Tyrone Wilkinson

6 Appendix

1 Introduction

What is the best way to fit several objects together to take up the least amount of space? How can the elements of a system be arranged if only certain parts of each element can be touching. Given a set of objects can they be arranged in such a way that there are no gaps? These types of problems, and many more, can be solved by a Blox program.

Blox is an object oriented programming language designed to facilitate the algorithmic creation of three-dimensional objects. The basic idea in Blox is that any three-dimensional object can be represented as a 3D array of blocks. Using Blox the developer can create and manipulate these 3D arrays to represent complicated 3D objects. Built-in functions in the language help the developer solve structural problems by joining separate parts together, according to specified rules, to create a desired object. The developer can use Blox to convert any object they create into an Additive Manufacturing File (AMF), which can then be viewed using 3D viewing software or printed with a 3D printer.

The AMF format was chosen as the best possible output for several reasons. First and perhaps most importantly, AMF introduced a host of new features such as allowing for direct control over an object's color(s), texture(s), and composition(s), while managing to be backwards compatible with the aging STereoLithography (STL) file format. It was also designed to be non-proprietary, which makes it a better choice than the similarly-formatted 3D Manufacturing Format (3MF) that Microsoft introduced and heavily promoted not long after AMF's approval. The extensive capabilities of AMF increases the potential of our language.

2 Language Reference Manual

2.1 Lexical Conventions

This section describes the lexical elements that make up Blox source code after preprocessing. These elements are called tokens. There are five types of tokens: , , , , and . White space, sometimes required to separate tokens, is also described in this chapter.

Identifiers

Identifiers are sequences of characters used for naming variables, functions, new data types, and preprocessor macros. You can include letters, decimal digits, and the underscore character ‘_’ in identifiers. Lowercase letters and uppercase letters are distinct, such that foo and FOO are two different identifiers.

Keywords

Keywords are special identifiers reserved for use as part of the programming language itself. You cannot use them for any other purpose. Here is a list of keywords recognized in Blox:

if else for while break continue return void null
int float bool true false string Frame Face
print Convert Build Join

Constants

A constant is a literal numeric or character value, such as 5 or 'm'. All constants are of a particular data type; you must use type casting to explicitly specify the type of a constant. There are integer constants, real number constants, character constants, and string constants.

Integer Constants

An integer constant is a sequence of digits assumed to be in base 10, so no prefixes are used.

```
0123
-45
9
```

Real Number Constants

A real number constant is a value that represents a fractional (floating point) number. It consists of a sequence of digits which represents the integer (or "whole") part of the number, a decimal point, and a sequence of digits which represents the fractional part. Either the integer part or the fractional part may be omitted, but not both. The exponent can be either positive or negative. Real number constants cannot be followed by e or E and an integer exponent.

```
4.2
.5
0.88
```

String Constants

A string constant is a sequence of zero or more characters, digits, and escape sequences enclosed within double quotation marks. A string constant is of type "array of characters". All string constants contain a null termination character (\0) as their last character. Strings are stored as arrays of characters, with no inherent size attribute. The null termination character allows string-processing functions know where the string ends.

Separators

A separator separates tokens. White space (see subsection) is a separator, but it is not a token. The other separators are all single-character tokens themselves:

( ) < > , ;

White Space

White space is the collective term used for several characters: the space character, the tab character, the newline character and the horizontal tab character. White space is ignored, and is therefore optional, except when it is used to separate tokens.

Comments

The characters **/*** introduce a comment, which is terminated with the characters ***/**. Block comments are supported, but nested comments are not.

2.2 Operators

Blox supports three types of operators.

1. Assignment operators
2. Comparison operators
3. Logical operators.

Assignment operators store values in variables. The standard assignment operator **=** simply stores the value of its right operand in the variable specified by its left operand. As with all assignment operators, the left operand cannot be a literal or constant value.

Comparison operators are used to compare two objects of the same type. Expressions created with comparison operators have a result of type bool, either true or false. Comparison operators can be used with the basic primitive types as well as Frames. The equal-to operator **==** tests two Frames for structural equality, are they composed of the same number of blocks in the same three dimensional arrangement. The result is true if the Frames are equal, and false if the Frames are not equal. The not-equal-to operator **!=** tests two Frames for structural inequality. The result is true if the Frames are not equal, and false if the Frames are equal. The dot-equal-to operator, called block equality **.=** is only for Frames, and determines whether or not two Frames have the same number of Blocks. The result is of type bool, true if the Frames have equal number of Blocks, and false if the Frames have an unequal number of Blocks. Beyond equality, inequality and block-equality, there are comparison operators greater than, less than, greater than or equal to, and less than or equal to.

Logical operators test the truth value of a pair of operands. All non-zero expressions are considered true, while any expression evaluating to zero is considered false.

Operator	**Name**	**Example**
=	Assign	x = 6; /* The value of variable x is now 6 */
+	Addition	y = x + 4; /* The value of y is now 10 */
-	Subtraction	z = y - x; /* The value of z is now 4 */
*	Multiplication	a = x * y; /* The value of a is now 24 */
/	Division	b = y / 5; /* The value of b is now 2 */
%	Modulo	d = c % x; /* The value of d is now 4 */
&&	Logical AND	expr1 && expr2 /* Returns true only if both expr are true, otherwise it returns false */
||	Logical OR	expr1 || expr2 /* Returns true if one or both expr are true, if both are false it returns false */
!	Logical NOT	!expr1 /* Returns true if expr1 is false, returns false if expr1

		is true */
==	Equal To	z == d /* Returns true */ x == y /* Returns false */
!=	Not Equal To	z != d /* Returns false */ x != y /* Returns true */
>	Greater Than	x > y /* Returns false */ a > b /* Returns true */
>=	Greater Than or Equal To	a >= b /* Returns true */ z >= d /* Returns true */
<	Less Than	x < y /* Returns true */ a < b /* Returns false */
<=	Less Than or Equal To	x <= y /* Returns true */ z <= d /* Returns true */
[ ]	Access Array Element	array[0] /* Returns first element in array */
.	Access Object Member	object.mem /* Returns member mem in object */

Precedence of Operators

```
[ ]   .
!
*   /   %
+   -
>   >=   <   <=
==   !=
&&
||
=
```

2.3 Data Types

Primitive Types

Primitive types are predefined in the Blox language as the most basic data types available, and are named by their keywords.

int from -2147483648 to 2147483647, inclusive

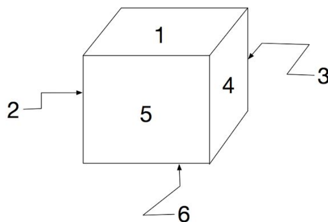
float from 3.402,823,5 E+38 to 1.4 E-45, inclusive
bool **true** or **false**
string “this is a string“

Language Specific Types

There are two language specific data types in Blox: Frame and Face.

Frame

A Frame can be used to represent any object in three dimensional space. It is a record data type comprised of its dimensions and an array of Blocks , whose indices represent three dimensional coordinates. A Block is the fundamental abstract object of the language. Although the programmer cannot directly manipulate blocks, the whole understanding of Frames is built upon Blocks. A block is a cube in three dimensional space, represented by a 6 index array of bools. Each bool represents a face of the block, and keeps track of whether the face is open/available (true), or taken/unavailable (false). A conceptual illustration is provided below:



	Side	Name	Identifier
1.	Positive y	North	“N”
2.	Negative x	West	“W”
3.	Negative z	Back	“B”
4.	Positive x	East	“E”
5.	Positive z	Front	“F”
6.	Negative y	South	“S”

Declaring and Initializing Frames

The programmer declares a Frame and initializes its contents by using the Frame keyword, specifying its dimensions and its name. Declaration and initialization must be done together.

Syntax

```
Frame<                                    >       ;
```

Example

```
Frame<2,2,1> a;
Frame<50,1,50> b;
```

Upon declaration, a frame is a simple rectangular prism with the dimensions specified at declaration. However, by performing Joins or Builds, the programmer can create complex Frames that have any type of irregular shape by joining simple Frames together. Frames are never disjoint, and cannot overlap.

Face

The Face data type is used to specify a certain point on a Frame. It is a record data type that comprises of a 3-tuple , which represents coordinates, and , which specifies a side of the block at coordinates . Face objects are used in the Join and Build functions to specify where to join two frames together. Face objects are usually created for a specific Frame, but one Face can be applied with multiple Frames effectively.

Syntax

```
Face<  >  ;
```

Example

```
Face<1,2,3,N> y;
```

Collection Types

Array

Arrays are declared by specifying the data type for its elements, the number of elements it can store enclosed within a set of brackets, and then its name. Array sizes are constant and indices start at value 0. Arrays can be initialized during or after declaration by referencing the index.

Syntax

```
  [  ]  ;
```

Example

```
int[2] x;
Face<1,2,3,N>[5] y;
Frame<5,5,5>[20] z;
```

2.4 Expressions

An an expression is any legal combination of symbols that represents a value. Expressions can include separators, operators, variables, constants, and function calls. When an expression has subexpressions, the innermost expressions are evaluated first.

Example

```
( x + ( ( 5 * 74 ) / 37 ) - 4 )
```

In the above example, 5 * 74 evaluates to 370. Then, 37 is divided from 370, resulting in 10. 4 is subtracted from 10, resulting in 6, which is finally added to x. The outermost parentheses are not required.

2.5 Scope

Blox is a block-structured language, meaning the lexical scope of variables do not extend beyond the pair of curly braces in which they are declared.

Example

```
{
    x = 5;
    {
        x = 1;
        y = 2;
    }
    print(x);        /* the value of x is 5 */
}
```

2.6 Statements

All statements must end with the semicolon **;** character.

Block statements

Syntax

```
{statement1; statement2; statement3; ... }
```

Example

```
{
    x = 1;
    y = 2;
}
```

Conditional statements

Syntax

```
if (expression) {statements}
if (expression) {statements} else {statements}
if (expression) {statements} else if (expression) {statements} else {statements}
```

Example

```
if (x == y) {return x+1;}
if (x == y) {return x+1;} else {return x-1;}
if (x == y) {return x+1;} else if (x < 1) {return x-1;} else {return 0;}
```

Loop statements

For loop

Syntax

```
for (expression; expression; expression) { statements }
```

Example

```
int i;
for (i = 0; i < 10; i = i + 1)
{
	print("Hello ");
	print("World!");
}
```

While loop

Syntax

```
while (expression) { statements }
```

Example

```
int x = 0;
while (x < 10)
{
	print("Hello ");
	print("World!");
	x++;
}
```

2.7 Functions

A function returns either a data type or void. A function can be used as an expression itself.

Declaration

Syntax

```
( ) {}
```

Example

```
int add(int x, int y)
{
	int a = x + y;
```

```
        return a;
    }
```

Built-in Functions

Join

Join is a built in function that connects two Frames at specific locations to return a new Frame that represents the result of the Join.

Syntax

```
Frame        = Join (                         );
```

Example

```
Frame<1,1,1> x;
Frame<1,1,1> y;
Face<1,1,1,E> a;
Face<1,1,1,W> b;
Frame C = Join(x, a, y, b);     /* C is now a 2x1x1 Frame */
```

Join will throw several different errors if the user specifies a Join that isn't possible:

Face_Taken: One of the Faces specified is already taken
Block_Overlap: The Join would cause the two Frames to overlap
Invalid_Face: The face-identifier of one of the Faces specified isn't E, W, N, S, F, or B
Opposite_Face: The Faces specified are not opposites: E-W, N-S, or F-B
Invalid_Block: One of the Faces specifies an empty index of the Frame
Block_Out_Of_Bounds: One of the Faces specifies an index outside of the Frame

Build

Build takes two Frames and an array of Faces for each frame and returns an array of all possible Frames made by joining the two original Frames at the specified faces. If the Face array is empty for either Frame the algorithm assumes all open Faces as possible Join locations, and returns all possible results. Before returning, Build removes any duplicates or empty Frames.

Syntax

```
Frame<1,1,1>[]        = Build(                              );
```

Example

```
Frame<1,1,1> x;
Frame<1,1,1> y;
Frame<1,1,1>[] C = Build(x, [], y, []);
/* C is now an array of three frames: 2x1x1, 1x2x1, and 1x1x2 */
```

Build doesn't throw errors based on being passed arguments that cause invalid Joins, it simply skips over the result of such Joins. Therefore the programmer can be confident that all the Frames returned by Build are valid Frames.

Convert

Convert takes a single frame and converts it to an AMF file, which it saves in the current working directory. Convert returns 1 on success, 0 otherwise.

Syntax

```
Convert(          ;
```

Example

```
Frame<3,3,3> x;
Convert(x);
```

print

print prints primitive data types to stdout. It returns 1 on success, 0 on error. print is useful for displaying results of functions and monitoring the progress of the program.

Syntax

```
print(     ;
```

Example

```
int  x = 42;
print(x);                    /* 42 is printed to stdout */
```

3 Architecture

3.1 Blox Input

The source file to be compiled is a .blox file that adheres to the standards of the language as documented in the Language Reference Manual.

3.2 AST

The abstract syntax tree is generated by the parser and represents the overall structure of the program. All phases proceeding lexical analysis utilize the AST.

3.3 Scanner

The scanner handles the lexical analysis of the .blox file. It separates the program into tokens which include keywords, constant literals, and operators. Whitespace and comments are discarded. Illegal character combinations are caught here.

3.4 Parser

The parser performs syntactic analysis on the tokenized code. It analyzes the code for structure and verifies the syntax based on the defined Context-free Grammar (CFG)--a CFG is a set of rules that allow a string of tokens to transform into another token. The program is then represented as a tree of transformations from the symbol Program to the physical tokens, or terminals.

3.5 Analyzer

Semantic analysis is done by the analyzer, which involves analyzing the parsed code for meaning in order to determine whether it follows the rules of the language. Undeclared variables, duplicate variables, type mismatches, and the like are reported here. The scoping rules and variable declarations are checked here as well.

3.6 Executor

The executor houses the built-in functions and which the programmer can use to manipulate Frames, along with code that checks the validity of the programmer's usage of those functions.

3.7 Generator

Code generation is managed here. Every Frame the programmer calls on is converted into its associated AMF file.

3.8 Generated Code

The generated AMF file(s) based on the converted Frame(s) in the source code can be viewed, manipulated, and/or immediately printed from any compatible software.

4 Project Plan

4.1 Group-Member Responsibilities

The original Blox group consisted of four members, with the following responsibilities:

(DROPPED)	- Manager
Naeem Bhatti	- Language Guru
Tyrone Wilkinson	- System Architect
Jonathan Voss	- Tester

Unfortunately, a month into the project the old manager dropped the class. With three members remaining, coding and debugging was split among all group members, each focussing on a particular section of the compiler.

4.2 Project Goals

The goal of the project was to create a fully functional programming language that could create, manipulate, and convert to amf a representation of any 3D object, as well as handle C style arithmetic expressions and functions.

The intended results of compiling a Blox file were:

- An AMF file for any Frame on which Convert is called
- Any specified execution or results printed to stdout

Planned compiler options included:

- Limiting the maximum number of AMF files produced
- Limiting maximum runtime of a program
- Running commands to open generated AMF files with compatible software programs

Blox was designed with several applications in mind:

- Constraint Problems: Given a set of elements and a set of constraints on their interaction with each other, Blox would be used to find possible solutions to the system.
- Min/Max Volume, Surface Area, or Dimensions: Given a set of elements Blox would be able to find solutions that resulted in either the minimum or maximum for such measurements as volume, surface area, or a certain dimension.
- AMF file creation: Blox simplifies the creation of any AMF file, especially those that represent large geometric patterns.
- 3D Object Representation: Using Blox it would be possible to generate large libraries of basic 3D objects, which could in turn be used to create complex structures.

4.3 Challenges Faced

Through the course of the semester, the group ran into many challenges, and ultimately failed to meet the goals of the project. In the early phase of the project, the group had a hard time settling on a language idea. Once a basic proposal was agreed upon, a significant amount of time reasoning about the logic behind the language, how data types would be represented, how functions would manipulate the data types, and what the ultimate purpose of the language would be. Later in the project, the group began to realize how intensive allowing full manipulation of objects in three dimensions was. The planned functionality had to be cut down in several ways, including removing the ability to represent curves or any angle besides 90 degrees, to allow for rotations in orientation, to remove blocks from a Frame, or for each block to be assigned an individual color.

The main obstacle that the group faced was getting the various parts of the compiler to work together. In incremental testing, parts of the Scanner/Parser/Ast, functions that provide the logic of the compiler, and final code generation worked as intended. However getting everything to fit together without errors proved to be much harder than the group originally anticipated. This necessarily forced a reevaluation on the entire implementation since compiling down to AMF meant that the builtin functions and all of the source code would actually have to be executed in the compiler, which stood in sharp contrast to Micro C and the majority of projects from past semesters. It took some time for the group to: 1. Reconceptualize a language which had the basic functionality of any standard programming language yet incorporated the features that were desired, and 2. Design it so that its execution was integrated into its compilation. Unfortunately, it was rather late in the semester when the group once again solidified its vision of the language and identified the optimal method of actualizing it. Ultimately, the group was unable to complete a fully functional compiler to connect the functioning front-end, Generator, and AMF Converter, in a way that would allow for the necessary internal execution of the file.

5 Lessons Learned

5.1 Naeem Bhatti

Blox was just as much of a lesson on teamwork than it was was on writing and learning the inner workings of a compiler. The most challenging aspect of this semester was finding a way to instil a sense of urgency. Having a team member drop 3/4's of the way into the semester significantly hurt the group overall, however this was trumped by individual shortcomings. My advice to future students would be to heed the reviews and choose your team wisely. Finding motivated and capable teammates will have the biggest impact on the outcome of your language.

5.2 Jonathan Voss

From the beginning of the semester, I think that one thing we could have improved as a group was to have a well defined plan for successfully completing our project. For the first half of the semester there was little communication among the group and it took us a long time to all get on the same page. In future projects that have many complicated parts that need to fit together seamlessly, I will focus on creating a plan that includes all the necessary information for any member of the group I'm working with to have a complete understanding of the task at hand. Another mistake that I made personally in this project was to underestimate the sheer amount of work that would be required to reach our goals. If I would have worked to understand all the aspects of the project early on, I would have been able to work more effectively throughout the whole process. On a more specific note, I learned a lot about ocaml and how files are compiled. At the beginning of the semester I had no experience with ocaml or with what all goes into writing a coding language. Now I would like to consider myself fairly comfortable using ocaml, and although we failed to complete our compiler, I definitely gained a good understanding about the process. Given more time, I would be interested in further pursuing a project like Blox. If I had a second chance at this project I would really work to learn ocaml as early as possible, I feel like my progress was often delayed because I was wasting time trying to figure out how put the code I needed in the right format.

5.3 Tyrone Wilkinson

The primary lesson learned during this experience is the importance of fully developing a chosen concept as early as possible in order to determine its feasibility and gain a shared understanding of the work required to bring it to fruition. Once this has been achieved, all that remains is the distribution of responsibilities among team members so that the obstacles to overcome become mostly rooted in the team, obstacles such as personal diligence in completing his or her tasks and effective communication between members, rather than in the concept itself. Blox was refreshing and serviceable but perhaps it was these aspects that contributed to both our shared and unique miscalculations and underestimations, and to the numerous trials experienced, again both shared and unique, during the attempt to actualize our idea.

6 Appendix

ast.ml

```
(* binary operators *)
type op =
  | And   | Or  | Mod
  | Add   | Sub | Mult | Div
  | Equal | Neq | Less | Leq | Greater | Geq | FrameEq

(* unary operators *)
type uop = Neg | Not

(* if form is Face<x,y,z,d> A; then id = "";
   if form is Face A = B;      then id = A  *)

(* block face identifier *)
type face_id = {
  dim   : int * int * int;
  face  : string;
  fc_id : string
}

(* actual block *)
type blck = {
  faces : bool array;
}

(* if form is Frame<x,y,z> A; then id = "";
   if form is Frame A = B;    then id = A  *)

(* actual frame *)
type frame = {
  x : int;
  y : int;
  z : int;
  blocks : blck array;
  fr_id  : string
}

(* All types *)
type dtype =
  | Int | Bool | Float | String | Void
  | Frame of frame
  | FaceId of face_id
```

```
  | Array of dtype * int * string

(* built-in function call parameters *)
type join  = frame * face_id * frame * face_id
type build = frame * face_id array * frame * face_id array

(* variable declarations *)
type var_decl = dtype * string

(* expressions *)
type expr =
  | Id of string
  | Lit_Int of int
  | Lit_Flt of float
  | Lit_Str of string
  | Lit_Bool of bool
  | Assign of string * expr
  | Fr_assign of string * expr
  | Fc_assign of string * expr
  | Var_assign of dtype * string * expr
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Call of string * expr list
  | Null
  | Noexpr

(* variable assignments *)
type var_assign = dtype * string * expr

(* statements *)
type stmt =
  | Block of stmt list
  | Expr of expr
  | Join of join
  | Build of build
  | Print of expr
  | Array of dtype * int * string
  | Convert of frame
  | Var_decl of var_decl
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | Break
  | Continue
```

```
(* function declaration *)
type func_decl = {
  typ     : dtype;
  fname   : string;
  formals : var_decl list;
  body    : stmt list;
}

(* frame assignment - might need to be frame * frame *)
type fr_assign = string * string

(* face assignment - might need to be frame * frame *)
type fc_assign = string * string

(* gloabls is a combination of var & frame declarations and assignments *)
type globals = {
  var_decls  : var_decl list;
  var_assgns : var_assign list;
  fr_assgns  : fr_assign list;
  fc_assgns  : fc_assign list;
}

(* a blox program is a tuple of globals and function declarations *)
type program = globals list * func_decl list

(* print binary operators *)
let string_of_op = function
  | Add     -> "+"
  | Sub     -> "-"
  | Mult    -> "*"
  | Div     -> "/"
  | Equal   -> "=="
  | Neq     -> "!="
  | Less    -> "<"
  | Leq     -> "<="
  | Greater -> ">"
  | Geq     -> ">="
  | And     -> "&&"
  | Or      -> "||"
  | FrameEq -> ".="
  | Mod     -> "%"

(* print unary operators *)
let string_of_uop = function
  | Neg     -> "-"
  | Not     -> "!"

let string_of_dim (x,y,z) =
```

```
  string_of_int x ^ "," ^
  string_of_int y ^ "," ^
  string_of_int z

let rec get_type = function
  | Int          -> Int
  | Bool         -> Bool
  | String       -> String
  | Float        -> Float
  | Frame(fr)    -> Frame(fr)
  | FaceId(fc)   -> FaceId(fc)
  | Void         -> Void
  | Array(x,y,z) -> Array(x,y,z)

(* print datatypes *)
let rec string_of_dtype = function
  | Int          -> "int"
  | Bool         -> "bool"
  | String       -> "string"
  | Float        -> "float"
  | Frame(fr)    -> "Frame" ^ "<" ^ string_of_dim (fr.x, fr.y, fr.z) ^ ">"
  | FaceId(fc)   -> "Face"  ^ "<" ^ string_of_dim fc.dim ^ "," ^ fc.face ^ ">"
  | Void         -> "void"
  | Array(x,y,z) -> string_of_dtype x ^ "[" ^ string_of_int y ^ "] " ^ z

(* print expressions *)
let rec string_of_expr = function
  | Lit_Int(x)        -> string_of_int x
  | Lit_Flt(x)        -> string_of_float x
  | Lit_Str(x)        -> x
  | Id(x)             -> x
  | Lit_Bool(true)    -> "true"
  | Lit_Bool(false)   -> "false"
  | Assign(x,y)       -> x ^ " = " ^ string_of_expr y ^ ";"
  | Fr_assign(x,y)    -> "Frame " ^ x ^ " = " ^ string_of_expr y ^ ";"
  | Fc_assign(x,y)    -> "Face " ^ x ^ " = " ^ string_of_expr y ^ ";"
  | Var_assign(x,y,z) -> string_of_dtype x ^ " " ^ y ^ " = " ^ string_of_expr z ^ ";"
  | Null              -> "null"
  | Binop(e1,o,e2)    -> string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^
string_of_expr e2
  | Unop(o,e)         -> string_of_uop o   ^ string_of_expr e
  | Call(f,el)        -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^
")"
  | Noexpr            -> ""

(* print variable declarations *)
let string_of_var_decl (x,y) =
  string_of_dtype x ^ " " ^ y ^ ";"
```

```
(* print block face identifiers *)
let string_of_face_id (w,x,y,z) =
  "(" ^ string_of_int w ^ ", " ^
        string_of_int x ^ ", " ^
        string_of_int y ^ ", " ^
        z ^ ")"

(* print Join function arguments *)
let string_of_join_args (w,x,y,z) =
  w.fr_id ^ "," ^ x.fc_id ^ "," ^ y.fr_id ^ "," ^ z.fc_id

(* print block face identifiers - not currently used *)
let string_of_face_id_list faceid =
  "(" ^ string_of_dim faceid.dim ^ "," ^ faceid.face ^ ")"

(* print Build function arguments - will substitute the actual face values in here *)
let string_of_build_args (w,x,y,z) =
  let alist = (Array.to_list x) in
  let blist = (Array.to_list z) in
  w.fr_id ^ "," ^ (String.concat "," (List.map (fun f -> f.fc_id) alist)) ^ "," ^
  y.fr_id ^ "," ^ (String.concat "," (List.map (fun f -> f.fc_id) blist))

(* print statements *)
let rec string_of_stmt = function
  | Var_decl(x,y)     -> string_of_var_decl (x,y) ^ "\n"
  | Expr(expr)        -> string_of_expr expr      ^ "\n"
  | Join(w,x,y,z)     -> "Join("  ^ string_of_join_args  (w,x,y,z) ^ ");\n"
  | Build(w,x,y,z)    -> "Build(" ^ string_of_build_args (w,x,y,z) ^ ");\n"
  | Array(x,y,z)      -> string_of_dtype x ^ "[" ^ string_of_int y ^ "] " ^ z ^";\n"
  | Print(e)          -> "print(" ^ string_of_expr e ^ ");\n"
  | Convert(fr)       -> "Convert(" ^ fr.fr_id ^ ");\n"
  | Break             -> "break;\n"
  | Continue          -> "continue;\n"
  | Return(expr)      -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e,s,Block([])) -> "if ("    ^ string_of_expr e    ^ ")\n" ^ string_of_stmt s
  | If(e,s1,s2)       -> "if ("    ^ string_of_expr e    ^ ")\n" ^ string_of_stmt s1 ^
                         "else\n"  ^ string_of_stmt s2
  | For(e1,e2,e3,s)   -> "for ("   ^ string_of_expr e1   ^ "; " ^ string_of_expr e2 ^
                         "; "      ^ string_of_expr e3   ^ ") "  ^ string_of_stmt s
  | While(e,s)        -> "while (" ^ string_of_expr e    ^ ") "  ^ string_of_stmt s
  | Block(stmts)      -> "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^
"}\n"

(* print variable assignments *)
let string_of_vassign (t,id,exp) =
  string_of_dtype t ^ " " ^ id ^ " = " ^
  string_of_expr exp ^ ";\n"
```

```
(* print frame assignments *)
let string_of_frassign (fn1,fn2) =
  "Frame " ^ fn1 ^ " = " ^ fn2 ^ ";"

(* print face assignments *)
let string_of_fcassign (fn1,fn2) =
  "Face " ^ fn1 ^ " = " ^ fn2 ^ ";"

(* print function declarations *)
let string_of_func_decl fd =
  string_of_dtype fd.typ ^ " " ^ fd.fname ^ "(" ^
  String.concat ", " (List.map snd fd.formals)         ^ ")\n{\n" ^
  String.concat ""   (List.map string_of_stmt fd.body) ^ "}\n"

(* print globals *)
let string_of_globals glob =
  String.concat "" (List.map string_of_var_decl glob.var_decls) ^
  String.concat "" (List.map string_of_vassign glob.var_assgns) ^
  String.concat "" (List.map string_of_frassign glob.fr_assgns) ^
  String.concat "" (List.map string_of_fcassign glob.fc_assgns) ^"\n"

(* print blox program *)
let string_of_program (globals,funcs) =
  String.concat ""   (List.rev (List.map string_of_globals globals)) ^ "\n" ^
  String.concat "\n" (List.rev (List.map string_of_func_decl funcs))
```

scanner.mll

```
{ open Parser }

rule token = parse
  | [' ' '\t' '\r' '\n'] { token lexbuf   }    (* Whitespace *)
  | "/* "                { comment lexbuf }    (* Comments   *)
  | '='                  { ASSIGN   }
  | ','                  { COMMA    }
  | ';'                  { SEMI     }
  | '{'                  { LCURL    }
  | '}'                  { RCURL    }
  | '('                  { LPAREN   }
  | ')'                  { RPAREN   }
  | '['                  { LBRACK   }
  | ']'                  { RBRACK   }
  | '+'                  { PLUS     }
  | '-'                  { MINUS    }
  | '*'                  { TIMES    }
  | '/'                  { DIVIDE   }
```

```
  | '%'                   { MOD      }
  | '!'                   { NOT      }
  | '.'                   { DOT      }
  | '<'                   { LT       }
  | '>'                   { GT       }
  | "!="                  { NEQ      }
  | "=="                  { EQ       }
  | "<="                  { LEQ      }
  | ">="                  { GEQ      }
  | ".="                  { FRAMEEQ  }
  | "&&"                  { AND      }
  | "||"                  { OR       }
  | "if"                  { IF       }
  | "else"                { ELSE     }
  | "for"                 { FOR      }
  | "while"               { WHILE    }
  | "return"              { RETURN   }
  | "break"               { BREAK    }
  | "continue"            { CONTINUE }
  | "void"                { VOID     }
  | "null"                { NULL     }
  | "int"                 { INT      }
  | "bool"                { BOOL     }
  | "true"                { TRUE     }
  | "false"               { FALSE    }
  | "string"              { STRING   }
  | "float"               { FLOAT    }
  | "print"               { PRINT    }
  | "Convert"             { CONVERT  }
  | "Build"               { BUILD    }
  | "Join"                { JOIN     }
  | "Face"                { FACE     }
  | "Frame"               { FRAME    }
  | ['0'-'9']+ as lxm    { LIT_INT(int_of_string lxm) }
  | ['0'-'9']+ '.' ['0'-'9']+ as lxm { LIT_FLT(float_of_string lxm) }
  | ['\"'] [^'\"']* ['\"'] as lxm { LIT_STR(lxm) }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'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

```
%{ open Ast %}
```

```
%token ASSIGN COMMA SEMI
%token LCURL RCURL LPAREN RPAREN LBRACK RBRACK
%token PLUS MINUS TIMES DIVIDE MOD
%token NOT DOT
%token LT GT EQ NEQ LEQ GEQ FRAMEEQ AND OR
%token IF ELSE FOR WHILE RETURN BREAK CONTINUE
%token VOID INT BOOL STRING FLOAT
%token TRUE FALSE NULL EOF
%token PRINT BUILD JOIN FRAME SET FACE CONVERT
%token <string> ID
%token <string> LIT_STR
%token <float> LIT_FLT
%token <int> LIT_INT

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

%start program
%type <Ast.program> program

%%
program:
  decls EOF { $1 }

decls:
  |/* nothing */    { ([], [] )}
  | decls globals   { ($2 :: fst $1), snd $1 }
  | decls func_decl { fst $1, ($2 :: snd $1) }

dtype:
  | INT      { Int    }
  | FLOAT    { Float  }
  | BOOL     { Bool   }
  | STRING   { String }
  | VOID     { Void   }
  | FRAME frame_decl { Frame($2)  }
  | FACE face_decl   { FaceId($2) }
  | dtype LBRACK LIT_INT RBRACK ID { Array($1, $3, $5) }

face_decl:
```

```
  LT LIT_INT COMMA LIT_INT COMMA LIT_INT COMMA ID GT
    { { dim  = ($2,$4,$6); face = $8; fc_id = ""} }

frame_decl:
  LT LIT_INT COMMA LIT_INT COMMA LIT_INT GT
    { { x = $2; y = $4; z = $6; fr_id = ""; blocks = [||] } }

globals:
  | dtype ID SEMI                    /* var decls   */
    { { var_decls  = [($1, $2)];
        var_assgns = [];
        fr_assgns  = [];
        fc_assgns  = []; } }
  | dtype ID ASSIGN expr SEMI        /* var assigns */
    { { var_decls  = [];
        var_assgns = [($1, $2, $4)];
        fr_assgns  = [];
        fc_assgns  = []; } }
  | FRAME ID ASSIGN ID SEMI          /* fr assigns  */
    { { var_decls  = [];
        var_assgns = [];
        fr_assgns  = [($2, $4)];
        fc_assgns  = []; } }
  | FACE ID ASSIGN ID SEMI           /* fc assigns  */
    { { var_decls  = [];
        var_assgns = [];
        fr_assgns  = [];
        fc_assgns  = [($2, $4)]; } }

func_decl:
  dtype ID LPAREN formals_opt RPAREN LCURL stmt_list RCURL
    { { typ     = $1;
        fname   = $2;
        formals = $4;
        body    = List.rev $7 } }

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

formal_list:
  | dtype ID  { [($1,$2)] }
  | formal_list COMMA dtype ID { ($3,$4) :: $1 }

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

```
stmt:
  | expr SEMI              { Expr($1)           }
  | BREAK SEMI             { Break              }
  | CONTINUE SEMI          { Continue           }
  | dtype ID SEMI          { Var_decl($1,$2)    }
  | LCURL stmt_list RCURL  { Block(List.rev $2) }
  | RETURN SEMI            { Return Noexpr      }
  | RETURN expr SEMI       { Return $2          }
  | PRINT LPAREN expr RPAREN SEMI       { Print($3)         }
  | dtype LBRACK LIT_INT RBRACK ID SEMI { Array($1, $3, $5) }
  | CONVERT LPAREN ID RPAREN SEMI
    { Convert({ x = 0; y = 0; z = 0; fr_id = $3; blocks = [||] }) }
  | JOIN LPAREN ID COMMA ID COMMA ID COMMA ID RPAREN SEMI
    { Join( { x = 0; y = 0; z = 0; fr_id = $3; blocks = [||] },
            { dim = (0,0,0); face = ""; fc_id = $5},
            { x = 0; y = 0; z = 0; fr_id = $7; blocks = [||] },
            { dim = (0,0,0); face = ""; fc_id = $9}) }
  | BUILD LPAREN ID COMMA ID COMMA ID COMMA ID RPAREN SEMI
    { Build({ x = 0; y = 0; z = 0; fr_id = $3; blocks = [||] },
             Array.of_list [{ dim = (0,0,0); face = "*"; fc_id = $5}],
            { x = 0; y = 0; z = 0; fr_id = $7; blocks = [||] },
             Array.of_list [{ dim = (0,0,0); face = "*"; fc_id = $9}]) }
  | 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 SEMI expr_opt RPAREN stmt
    { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt
    { While($3, $5) }

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

expr:
  | ID                     { Id($1)                }
  | LIT_INT                { Lit_Int($1)           }
  | LIT_FLT                { Lit_Flt($1)           }
  | LIT_STR                { Lit_Str($1)           }
  | TRUE                   { Lit_Bool(true)        }
  | FALSE                  { Lit_Bool(false)       }
  | ID ASSIGN expr         { Assign($1, $3)        }
  | FRAME ID ASSIGN expr   { Fr_assign($2, $4)     }
  | FACE ID ASSIGN expr    { Fc_assign($2, $4)     }
  | dtype ID ASSIGN expr   { Var_assign($1, $2, $4) }
  | 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 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)   }
  | expr AND    expr       { Binop($1, And,   $3)   }
  | expr OR     expr       { Binop($1, Or,    $3)   }
  | MINUS expr %prec NEG   { Unop(Neg, $2)          }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3)     }
  | LPAREN expr RPAREN           { $2               }

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

actuals_list:
  | expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

analyzer.ml

```
open Ast

module StringMap = Map.Make(String)

let analyze (globals, functions) =

  (* Raise an exception if the given list has a duplicate frames *)
  let report_duplicate exceptf list =
    let rec helper = function
      | n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
      |  _ :: t -> helper t
      | [] -> ()
    in helper (List.sort compare list)
  in

  (* Raise an exception if a given binding is to a void type *)
  let check_not_void exceptf = function
    | (Void, n) -> raise (Failure (exceptf n))
    | _ -> ()
  in

  (* Raise an excp if given rvalue type can't be assigned to the given lvalue type *)
  let check_assign lvaluet rvaluet err =
```

```
    if lvaluet == rvaluet then lvaluet
    else raise err
  in

  (* Returns the global var decl list   *)
  let get_var_decl globs =
    (List.map (fun (x,y) -> y) globs.var_decls)
  in

  (* Check for restricted function names *)
  if List.mem "print" (List.map (fun fd -> fd.fname) functions) then
    raise (Failure ("function print may not be defined"))
  else ();
  if List.mem "build" (List.map (fun fd -> fd.fname) functions) then
    raise (Failure ("function build may not be defined"))
  else ();
  if List.mem "join" (List.map (fun fd -> fd.fname) functions) then
    raise (Failure ("function join may not be defined"))
  else ();
  if List.mem "convert" (List.map (fun fd -> fd.fname) functions) then
    raise (Failure ("function convert may not be defined"))
  else ();

  (* Check for duplicate function names *)
  report_duplicate (fun n -> "duplicate function " ^ n) (List.map (fun fd -> fd.fname)
functions);

  (* Function declarations for built-in functions *)
  let built_in_decls =
    StringMap.add "Join"
      { typ     = Void;
        fname   = "Join";
        formals = [ (Frame ({ x = 0; y = 0; z = 0; blocks = [||]; fr_id = ""}), "A");
                    (FaceId({ dim = (0,0,0); face = ""; fc_id = ""}), "B");
                    (Frame ({ x = 0; y = 0; z = 0; blocks = [||]; fr_id = ""}), "C");
                    (FaceId({ dim = (0,0,0); face = ""; fc_id = ""}), "D"); ];
        body    = [] };
    StringMap.add "Build"
      { typ     = Void;
        fname   = "Build";
        formals = [ (Frame ({ x = 0; y = 0; z = 0; blocks = [||]; fr_id = ""}), "A");
                    (FaceId({ dim = (0,0,0); face = ""; fc_id = ""}), "B");
                    (Frame ({ x = 0; y = 0; z = 0; blocks = [||]; fr_id = ""}), "C");
                    (FaceId({ dim = (0,0,0); face = ""; fc_id = ""}), "D"); ];
        body    = [] };
    StringMap.add "Convert"
      { typ     = Void;
        fname   = "Convert";
```

```
        formals = [(Frame ({ x = 0; y = 0; z = 0; blocks = [||]; fr_id = ""}), "A")];
        body    = [] };
    StringMap.add "print"
      { typ     = Void;
        fname   = "print";
        formals = [(String, "s")];
        body    = [] }
    (StringMap.singleton "print_bool"
      { typ     = Void;
        fname   = "print_bool";
        formals = [(Bool, "b")];
        body    = [] });
    (StringMap.singleton "print_int"
      { typ     = Void;
        fname   = "print_int";
        formals = [(Int, "i")];
        body    = [] });
    (StringMap.singleton "print_flt"
      { typ     = Void;
        fname   = "print_flt";
        formals = [(Float, "f")];
        body    = [] });
  in

  (* Add built-in function declarations to map, mapping: func_name -> func_decl *)
  let function_decls =
    List.fold_left (fun map fdecl -> StringMap.add fdecl.fname fdecl map)
built_in_decls functions
  in

  (* Check for unrecognized functions *)
  let function_decl s = try StringMap.find s function_decls
    with Not_found -> raise (Failure ("unrecognized function " ^ s))
  in

  (* Check for main function *)
  let check_main_decl = try StringMap.find "main" function_decls
    with Not_found -> raise (Failure ("missing main() entry point"))
  in
  check_main_decl;

  let check_globals glob =
    report_duplicate (fun n -> "duplicate global var/frame/face decl" ^ n)
      (List.rev (get_var_decl glob))
  in
  List.iter check_globals globals;
```

executor.ml

```
open Ast

exception Face_Taken of string;;
exception Block_Overlap of string;;
exception Invalid_Face of string;;
exception Opposite_Face of string;;
exception Invalid_Block of string;;
exception Block_Out_Of_Bounds of string;;


(* Return a frame with the given dimensions *)
let faceCons x y z w n =
  let fc = {dim = (x, y, z); face = w; fc_id = n} in
  fc;;

(* Takes 1D index and corresponding frame and returns 3D coordinates*)
let getCoord i frm =
  let z_val = (i mod frm.z) in
  let y_val = (((i - z_val) / frm.z) mod frm.y) in
  let x_val = ((((i - z_val) / frm.z) - y_val) / frm.y) in
  (x_val, y_val, z_val);;


(* Takes face array index returns face string *)
let getFcStr i = match i with
  0 -> "E"
| 1 -> "W"
| 2 -> "N"
| 3 -> "S"
| 4 -> "F"
| 5 -> "B"
| _ -> raise (Invalid_Face "Face index out of bounds");;


(* faceCheck takes a 1D array of blocks and its 3D dimensions, it updates block
  faces as unavailable if they are joined to another block *)
let faceCheck a x y z=

  let gx elx =
    if Array.length elx.faces = 6 then(
      Array.set elx.faces 1 false;
      false)
    else true in
  let gy ely =
```

```
    if Array.length ely.faces = 6 then(
      Array.set ely.faces 3 false;
      false)
    else true in
  let gz elz =
    if Array.length elz.faces = 6 then(
      Array.set elz.faces 5 false;
      false)
    else true in

  let f i el =
    if Array.length el.faces = 6 then(
      Array.set el.faces 0 (if ((i + (y*z)) < (x*y*z)) then(
        gx(a.(i + y*z)))
      else true;);
      Array.set el.faces 2 (if (((i mod (y*z)) + z) < (y*z)) then(
        gy(a.(i + z)))
      else true;);
      Array.set el.faces 4 (if (((i mod z) + 1) < z) then(
        gz(a.(i + 1)))
      else true)) in

  Array.iteri f a;;


(* Checks arguments given to join function and returns coordinate shifts *)
let argCheck frameA fidA frameB fidB =

  let (ax, ay, az) = fidA.dim in
  let (bx, by, bz) = fidB.dim in
  let af = fidA.face in
  let bf = fidB.face in
  let aArray = Array.init (Array.length frameA.blocks) (fun i -> {faces = Array.copy
frameA.blocks.(i).faces}) in
  let ai = ((((ax * frameA.y) + ay) * frameA.z) + az) in
  let bArray = Array.init (Array.length frameB.blocks) (fun i -> {faces = Array.copy
frameB.blocks.(i).faces}) in
  let bi = ((((bx * frameB.y) + by) * frameB.z) + bz) in

  (* Check specified blocks are within array boundaries *)
  if (ax >= frameA.x) || (ay >= frameA.y) || (az >= frameA.z)
    then raise (Block_Out_Of_Bounds "Specified block for first frame is outside array
boundaries");
  if (bx >= frameB.x) || (by >= frameB.y) || (bz >= frameB.z)
    then raise (Block_Out_Of_Bounds "Specified block for second frame is outside array
boundaries");

  (* Check given block exists *)
```

```
  if not (Array.length frameA.blocks.(ai).faces = 6)
    then raise (Invalid_Block "Specified Block for first frame does not exist");
  if not (Array.length frameB.blocks.(bi).faces = 6)
    then raise (Invalid_Block "Specified Block for second frame does not exist");

  (* Check for valid faces *)
  if (af = "E") || (af = "W") || (af = "N") || (af = "S") || (af = "F") || (af = "B")
    then ignore()
  else raise (Invalid_Face "Specified face for first frame must be E, W, N, S, F, or
B");
  if (bf = "E") || (bf = "W") || (bf = "N") || (bf = "S") || (bf = "F") || (bf = "B")
    then ignore()
  else raise (Invalid_Face "Specified face for second frame must be E, W, N, S, F, or
B");

  let aface =
    (if af = "E" then
      aArray.(ai).faces.(0)
    else if af = "W" then
      aArray.(ai).faces.(1)
    else if af = "N" then
      aArray.(ai).faces.(2)
    else if af = "S" then
      aArray.(ai).faces.(3)
    else if af = "F" then
      aArray.(ai).faces.(4)
    else if af = "B" then
      aArray.(ai).faces.(5)
    else false) in

  let (bface, bx_shift, by_shift, bz_shift) =
    (if bf = "E" then (bArray.(bi).faces.(0), (ax - 1) - bx, ay - by, az - bz)
    else if bf = "W" then(bArray.(bi).faces.(1), (ax + 1) - bx, ay - by, az - bz)
    else if bf = "N" then(bArray.(bi).faces.(2), ax - bx, (ay - 1) - by, az - bz)
    else if bf = "S" then(bArray.(bi).faces.(3), ax - bx, (ay + 1) - by, az - bz)
    else if bf = "F" then(bArray.(bi).faces.(4), ax - bx, ay - by, (az - 1) - bz)
    else if bf = "B" then(bArray.(bi).faces.(5), ax - bx, ay - by, (az + 1) - bz)
    else (false, 0, 0, 0)) in

  (* check if frameA's block face is available *)
  if not(aface) then
    raise (Face_Taken "Specified face of block in first frame is unavailable");

  (* check if frameB's block face is available *)
  if not(bface) then
    raise (Face_Taken "Specified face of block in second frame is unavailable");

  (* check for opposite faces *)
```

```
  if (((af = "E") && not(bf = "W")) ||
      ((af = "W") && not(bf = "E"))) then
    raise (Opposite_Face "Must specify opposite faces");
  if (((af = "N") && not(bf = "S")) ||
      ((af = "S") && not(bf = "N"))) then
    raise (Opposite_Face "Must specify opposite faces");
  if (((af = "F") && not(bf = "B")) ||
      ((af = "B") && not(bf = "F"))) then
    raise (Opposite_Face "Must specify opposite faces");

  (* Determine shift values for A and B *)
  let (ax_shift, bx_shift) =
    (if bx_shift < 0 then (-bx_shift, 0) else (0, bx_shift)) in
  let (ay_shift, by_shift) =
    (if by_shift < 0 then (-by_shift, 0) else (0, by_shift)) in
  let (az_shift, bz_shift) =
    (if bz_shift < 0 then (-bz_shift, 0) else (0, bz_shift)) in

  (* Return shift values and copied arrays *)
  (ax_shift, ay_shift, az_shift, bx_shift, by_shift, bz_shift, aArray, bArray);;


(* Joins two frames *)
let join frameA fidA frameB fidB =

  let (ax_shift, ay_shift, az_shift, bx_shift, by_shift, bz_shift, aArray, bArray) =
argCheck frameA fidA frameB fidB in

  (* Determine size of new array *)
  let cx = (max (frameA.x + ax_shift) (frameB.x + bx_shift)) in
  let cy = (max (frameA.y + ay_shift) (frameB.y + by_shift)) in
  let cz = (max (frameA.z + az_shift) (frameB.z + bz_shift)) in

  (* Create new array of blocks *)
  let c = Array.init (cx * cy * cz) (fun _ -> let b = {faces = [||]} in b ) in

  (* Fill c with blocks from array A *)
  let f i el =
    if Array.length el.faces = 6 then(
      let (x_val, y_val, z_val) = getCoord i frameA in
      let cz_val = z_val + az_shift in
      let cy_val = y_val + ay_shift in
      let cx_val = x_val + ax_shift in
      let ci = ((((cx_val * cy) + cy_val) * cz) + cz_val) in
      Array.set c ci el) in
  Array.iteri f aArray;

  (* Fill c with blocks from array B *)
```

```
  let g i el =
    if Array.length el.faces = 6 then(
      let (x_val, y_val, z_val) = getCoord i frameB in
      let cz_val = z_val + bz_shift in
      let cy_val = y_val + by_shift in
      let cx_val = x_val + bx_shift in
      let ci = ((((cx_val * cy) + cy_val) * cz) + cz_val) in
      if (Array.length c.(ci).faces = 0) then(
        Array.set c ci el)
      else raise (Block_Overlap "The specified join causes overlap")) in
  Array.iteri g bArray;

  (* Run faceCheck *)
  faceCheck c cx cy cz;

  (* Create and return resulting frame C *)
  let frameC = {
  x = cx;
  y = cy;
  z = cz;
  blocks = c;
  fr_id = "Result";
  } in

  frameC;;


(* build takes two frames and an array of face ID's for each frame, returns
   all possible frames made by joining the two original frames at the specified
   faces. If the faceID array is empty for either frame the algorithm assumes
   all open faces as possible join locations *)
let build frameA faceArrA frameB faceArrB =

  (* Create an array large enough to hold the maximum number of possible results *)
  let returnArr = match (Array.length faceArrA, Array.length faceArrB) with
      (0, 0)  ->  Array.init (6*(Array.length frameA.blocks)*(Array.length
frameB.blocks)) (fun _ -> {x = 0; y = 0; z = 0; blocks = [||]; fr_id = ""})
    | (a, 0)  ->  Array.init (a*(Array.length frameA.blocks)) (fun _ -> {x = 0; y = 0;
z = 0; blocks = [||]; fr_id = ""})
    | (0, b)  ->  Array.init (b*(Array.length frameB.blocks)) (fun _ -> {x = 0; y = 0;
z = 0; blocks = [||]; fr_id = ""})
    | (a, b)  ->  Array.init (a*b) (fun _ -> {x = 0; y = 0; z = 0; blocks = [||]; fr_id
= ""}) in

  (* Return all faces in fLB that can be joined to face el *)
  let fndFcsB el fLB = match el.face with
    | "E" -> Array.map (fun x -> if (x.face = "W") then x else {dim = (0,0,0); face =
"Empty"; fc_id = ""}) fLB
```

```
    | "W" -> Array.map (fun x -> if (x.face = "E") then x else {dim = (0,0,0); face =
"Empty"; fc_id = ""}) fLB
    | "N" -> Array.map (fun x -> if (x.face = "S") then x else {dim = (0,0,0); face =
"Empty"; fc_id = ""}) fLB
    | "S" -> Array.map (fun x -> if (x.face = "N") then x else {dim = (0,0,0); face =
"Empty"; fc_id = ""}) fLB
    | "F" -> Array.map (fun x -> if (x.face = "B") then x else {dim = (0,0,0); face =
"Empty"; fc_id = ""}) fLB
    | "B" -> Array.map (fun x -> if (x.face = "F") then x else {dim = (0,0,0); face =
"Empty"; fc_id = ""}) fLB
    |  _  -> raise (Invalid_Face "A given face string for the first frame is not
formated as one of: E, W, N, S, F, B") in

  (* Return all available faces in frm *)
  let allFc frm =
    let allFcArr = Array.init ((Array.length frm.blocks) * 6) (fun _ -> {dim = (0,0,0);
face = "Empty"; fc_id = ""}) in
    let count = ref 0 in
    let finder1 i el =
      let finder2 j fel =
        if fel then(
          Array.set allFcArr !count {dim = (getCoord i frm); face = (getFcStr j); fc_id
= ""};
          incr count) in
      if Array.length el.faces = 6 then(
        Array.iteri finder2 el.faces) in
    Array.iteri finder1 frm.blocks;
    allFcArr in

  (* join frameA and frameB at every possible combination in flA and flB *)
  let joinAB fLA fLB =
    let count = ref 0 in
    let joiner1 elA =
      let joiner2 elB =
        if elB.face = "Empty" then ignore() else(
          try(
            Array.set returnArr !count (join frameA (elA) frameB (elB));
            incr count)
          with
          | Face_Taken x          -> ignore()
          | Block_Overlap x       -> ignore()
          | Invalid_Face x        -> ignore()
          | Opposite_Face x       -> ignore()
          | Invalid_Block x       -> ignore()
          | Block_Out_Of_Bounds x -> ignore())in
      if elA.face = "Empty" then ignore() else(Array.iter joiner2 (fndFcsB elA fLB)) in
    Array.iter joiner1 fLA in
```

```
  (* Call join on specified faces of frameA and frameB *)
  let num fLA fLB = match (Array.length fLA, Array.length fLB) with
    | (0, 0) -> joinAB (allFc frameA) (allFc frameB)
    | (x, 0) -> joinAB fLA (allFc frameB)
    | (0, y) -> joinAB (allFc frameA) fLB
    | (x, y) -> joinAB fLA fLB in

  num faceArrA faceArrB;

  (* Remove duplicate and empty frames from results *)
  let returnList = Array.to_list returnArr in
  let returnList = List.sort_uniq compare returnList in
  let returnList = if (List.hd returnList).x = 0 then List.tl returnList else
returnList in
  let returnArr = Array.of_list returnList in

  returnArr;;



  (* RETURN ONE FRAME VERSION OF BUILD *)
  (* build takes two frames and an array of face ID's for each frame, returns
   all possible frames made by joining the two original frames at the specified
   faces. If the faceID array is empty for either frame the algorithm assumes
   all open faces as possible join locations *)
let buildone frameA faceArrA frameB faceArrB returnstring =

  (* Create an array large enough to hold the maximum number of possible results *)
  let returnArr = match (Array.length faceArrA, Array.length faceArrB) with
      (0, 0)  ->  Array.init (6*(Array.length frameA.blocks)*(Array.length
frameB.blocks)) (fun _ -> {x = 0; y = 0; z = 0; blocks = [||]; fr_id = ""})
    | (a, 0)  ->  Array.init (a*(Array.length frameA.blocks)) (fun _ -> {x = 0; y = 0;
z = 0; blocks = [||]; fr_id = ""})
    | (0, b)  ->  Array.init (b*(Array.length frameB.blocks)) (fun _ -> {x = 0; y = 0;
z = 0; blocks = [||]; fr_id = ""})
    | (a, b)  ->  Array.init (a*b) (fun _ -> {x = 0; y = 0; z = 0; blocks = [||]; fr_id
= ""}) in

  (* Return all faces in fLB that can be joined to face el *)
  let fndFcsB el fLB = match el.face with
    | "E" -> Array.map (fun x -> if (x.face = "W") then x else {dim = (0,0,0); face =
"Empty"; fc_id = ""}) fLB
    | "W" -> Array.map (fun x -> if (x.face = "E") then x else {dim = (0,0,0); face =
"Empty"; fc_id = ""}) fLB
    | "N" -> Array.map (fun x -> if (x.face = "S") then x else {dim = (0,0,0); face =
"Empty"; fc_id = ""}) fLB
    | "S" -> Array.map (fun x -> if (x.face = "N") then x else {dim = (0,0,0); face =
"Empty"; fc_id = ""}) fLB
```

```
    | "F" -> Array.map (fun x -> if (x.face = "B") then x else {dim = (0,0,0); face =
"Empty"; fc_id = ""}) fLB
    | "B" -> Array.map (fun x -> if (x.face = "F") then x else {dim = (0,0,0); face =
"Empty"; fc_id = ""}) fLB
    |  _  -> raise (Invalid_Face "A given face string for the first frame is not
formated as one of: E, W, N, S, F, B") in

  (* Return all available faces in frm *)
  let allFc frm =
    let allFcArr = Array.init ((Array.length frm.blocks) * 6) (fun _ -> {dim = (0,0,0);
face = "Empty"; fc_id = ""}) in
    let count = ref 0 in
    let finder1 i el =
      let finder2 j fel =
        if fel then(
          Array.set allFcArr !count {dim = (getCoord i frm); face = (getFcStr j); fc_id
= ""};
          incr count) in
      if Array.length el.faces = 6 then(
        Array.iteri finder2 el.faces) in
    Array.iteri finder1 frm.blocks;
    allFcArr in

  (* join frameA and frameB at every possible combination in flA and flB *)
  let joinAB fLA fLB =
    let count = ref 0 in
    let joiner1 elA =
      let joiner2 elB =
        if elB.face = "Empty" then ignore() else(
          try(
            Array.set returnArr !count (join frameA (elA) frameB (elB));
            incr count)
          with
          | Face_Taken x          -> ignore()
          | Block_Overlap x       -> ignore()
          | Invalid_Face x        -> ignore()
          | Opposite_Face x       -> ignore()
          | Invalid_Block x       -> ignore()
          | Block_Out_Of_Bounds x -> ignore())in
      if elA.face = "Empty" then ignore() else(Array.iter joiner2 (fndFcsB elA fLB)) in
    Array.iter joiner1 fLA in

  (* Call join on specified faces of frameA and frameB *)
  let num fLA fLB = match (Array.length fLA, Array.length fLB) with
    | (0, 0) -> joinAB (allFc frameA) (allFc frameB)
    | (x, 0) -> joinAB fLA (allFc frameB)
    | (0, y) -> joinAB (allFc frameA) fLB
    | (x, y) -> joinAB fLA fLB in
```

```
(* Returns the smallest overall frame from the build *)
let smallest iA =
  let index = ref 0 in
  let xyz = ref 1000000 in
  let finder i el =
    if (el.x + el.y + el.z) < !xyz then(
      xyz := (el.x + el.y + el.z);
      index := i) in
  Array.iteri finder iA;
  iA.(!index) in

(* Returns the largest overall frame from the build *)
let largest iA =
  let index = ref 0 in
  let xyz = ref 0 in
  let finder i el =
    if (el.x + el.y + el.z) > !xyz then(
      xyz := (el.x + el.y + el.z);
      index := i) in
  Array.iteri finder iA;
  iA.(!index) in

(* Returns the frame with the smallest x dimension from the build *)
let smallestx iA =
  let index = ref 0 in
  let x = ref 1000000 in
  let finder i el =
    if el.x < !x then(
      x := el.x;
      index := i) in
  Array.iteri finder iA;
  iA.(!index) in

(* Returns the frame with the largest x dimension from the build *)
let largestx iA =
  let index = ref 0 in
  let x = ref 0 in
  let finder i el =
    if el.x > !x then(
      x := el.x;
      index := i) in
  Array.iteri finder iA;
  iA.(!index) in

(* Returns the frame with the smallest y dimension from the build *)
 let smallesty iA =
  let index = ref 0 in
```

```
  let y = ref 1000000 in
  let finder i el =
    if el.y < !y then(
      y := el.y;
      index := i) in
  Array.iteri finder iA;
  iA.(!index) in

(* Returns the frame with the largest y dimension from the build *)
 let largesty iA =
  let index = ref 0 in
  let y = ref 0 in
  let finder i el =
    if el.y > !y then(
      y := el.y;
      index := i) in
  Array.iteri finder iA;
  iA.(!index) in

(* Returns the frame with the smallest z dimension from the build *)
let smallestz iA =
  let index = ref 0 in
  let z = ref 1000000 in
  let finder i el =
    if el.z < !z then(
      z := el.z;
      index := i) in
  Array.iteri finder iA;
  iA.(!index) in

(* Returns the frame with the largest z dimension from the build *)
let largestz iA =
  let index = ref 0 in
  let z = ref 0 in
  let finder i el =
    if el.z > !z then(
      z := el.z;
      index := i) in
  Array.iteri finder iA;
  iA.(!index) in

(* Returns a random frame from the build *)
let random iA =
  let n = Random.int (Array.length iA) in
  Array.get iA n in

(* Matches build return specifier, calls function to return *)
let returner rs rA = match rs with
```

```
      "smallest"  ->  smallest rA
    | "largest"   ->  largest rA
    | "smallestx" ->  smallestx rA
    | "largestx"  ->  largestx rA
    | "smallesty" ->  smallesty rA
    | "largesty"  ->  largesty rA
    | "smallestz" ->  smallestz rA
    | "largestz"  ->  largestz rA
    | _           ->  random rA in

  num faceArrA faceArrB;

  (* Remove duplicate and empty frames from results *)
  let returnList = Array.to_list returnArr in
  let returnList = List.sort_uniq compare returnList in
  let returnList = if (List.hd returnList).x = 0 then List.tl returnList else
returnList in
  let returnArr = Array.of_list returnList in

  if (Array.length returnArr) > 0 then returner returnstring returnArr else {x = 0; y =
0; z = 0; blocks = [||]; fr_id = ""};;


(* Return a frame with the given dimensions *)
let frameCons x y z n =
  let arr = Array.init (x*y*z) (fun _ -> {faces = [|true;true;true;true;true;true|]})
in
  let frm = {x = x; y = y; z = z; blocks = arr; fr_id = n} in
  faceCheck frm.blocks x y z;
  frm;;

let execute (globals, functions) =
  print_endline "Add stack execution code here ...\n"
```

generator.ml

```
open Ast
open Printf

exception Face_Taken of string;;
exception Block_Overlap of string;;
exception Invalid_Face of string;;
exception Opposite_Face of string;;
exception Invalid_Block of string;;
exception Block_Out_Of_Bounds of string;;
```

```
let generate frm =
    let name = frm.fr_id ^ ".amf" in
    let oc = open_out name in
      let x = ref 0 in
      let y = ref 0 in
      let z = ref 0 in
      let z_val = ref 0 in
      let y_val = ref 0 in
      let x_val = ref 0 in
      let vertices = ref [|0; 1; 2; 3; 4; 5; 6; 7|] in
      let line = ref 8 in
      let top = "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n<amf>\n\t<object
id=\"1\">\n\t\t<mesh>" in
      let bottom = "\t\t</mesh>\n\t</object>\n</amf>" in
      let vertexstart = "\t\t\t<vertices>" in
      let vertexend = "\t\t\t</vertices>" in
      let vpart1 = "<vertex><coordinates><x>" in
      let vpart2 = "</x><y>" in
      let vpart3 = "</y><z>" in
      let vpart4 = "</z></coordinates></vertex>" in
      let trianglestart = "\t\t\t<volume>" in
      let triangleend = "\t\t\t</volume>" in
      let tpart1 = "<triangle><v1>" in
      let tpart2 = "</v1><v2>" in
      let tpart3 = "</v2><v3>" in
      let tpart4 = "</v3></triangle>" in

      let check = ref 0 in
      let displacement = ref 0 in
      let actlength = ref 0 in
      let length = ref (Array.length frm.blocks) in

      fprintf oc "%s\n" top;
      fprintf oc "%s\n" vertexstart;

      while (!check < !length) do(
        if ((Array.length frm.blocks.(!check).faces) = 6) then(
          z_val := (!check mod frm.z);
          y_val := (((!check - !z_val) / frm.z) mod frm.y);
          x_val := ((((!check - !z_val) / frm.z) - !y_val) / frm.y);

            while (!line > 0) do(
              if (!line > 4) then
                x := !x_val
              else(
                x := !x_val - 1
              );
              if (!line mod 4 = 0 || !line mod 4 = 3) then
```

```
            y := !y_val
          else(
            y := !y_val - 1
          );
          if (!line mod 2 = 0) then
            z := !z_val
          else(
            z := !z_val - 1
          );
          fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" vpart1 !x vpart2 !y vpart3 !z
vpart4;
          line := !line - 1
        )done;
          check := !check + 1;
          actlength := !actlength + 1;
          line := 8
      )
      else(
        check := !check + 1
      )
    )done;

    fprintf oc "%s\n" vertexend;
    fprintf oc "%s\n" trianglestart;
    check := 0;

    while (!actlength > !check) do(
      displacement := !check * 8;
      fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" tpart1 (!vertices.(0) + !displacement)
tpart2 (!vertices.(1) + !displacement) tpart3 (!vertices.(3) + !displacement) tpart4;
(* write each triangle *)
      fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" tpart1 (!vertices.(0) + !displacement)
tpart2 (!vertices.(2) + !displacement) tpart3 (!vertices.(3) + !displacement) tpart4;
(* write each triangle *)

      fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" tpart1 (!vertices.(5) + !displacement)
tpart2 (!vertices.(6) + !displacement) tpart3 (!vertices.(7) + !displacement) tpart4;
(* write each triangle *)
      fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" tpart1 (!vertices.(4) + !displacement)
tpart2 (!vertices.(5) + !displacement) tpart3 (!vertices.(6) + !displacement) tpart4;
(* write each triangle *)

      fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" tpart1 (!vertices.(0) + !displacement)
tpart2 (!vertices.(1) + !displacement) tpart3 (!vertices.(5) + !displacement) tpart4;
(* write each triangle *)
      fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" tpart1 (!vertices.(0) + !displacement)
tpart2 (!vertices.(4) + !displacement) tpart3 (!vertices.(5) + !displacement) tpart4;
(* write each triangle *)
```

```
        fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" tpart1 (!vertices.(2) + !displacement)
tpart2 (!vertices.(3) + !displacement) tpart3 (!vertices.(7) + !displacement) tpart4;
(* write each triangle *)
        fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" tpart1 (!vertices.(2) + !displacement)
tpart2 (!vertices.(6) + !displacement) tpart3 (!vertices.(7) + !displacement) tpart4;
(* write each triangle *)

        fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" tpart1 (!vertices.(0) + !displacement)
tpart2 (!vertices.(2) + !displacement) tpart3 (!vertices.(6) + !displacement) tpart4;
(* write each triangle *)
        fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" tpart1 (!vertices.(0) + !displacement)
tpart2 (!vertices.(4) + !displacement) tpart3 (!vertices.(6) + !displacement) tpart4;
(* write each triangle *)

        fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" tpart1 (!vertices.(1) + !displacement)
tpart2 (!vertices.(3) + !displacement) tpart3 (!vertices.(7) + !displacement) tpart4;
(* write each triangle *)
        fprintf oc "\t\t\t\t%s%d%s%d%s%d%s\n" tpart1 (!vertices.(1) + !displacement)
tpart2 (!vertices.(5) + !displacement) tpart3 (!vertices.(7) + !displacement) tpart4;
(* write each triangle *)

        check := !check + 1;
      )done;

      fprintf oc "%s\n" triangleend;
      fprintf oc "%s\n" bottom;
      close_out oc;;
```

Makefile

```
.PHONY: Create-Scanner Create-Parser Create-AST \
             Compile-Scanner \
             Compile-Parser Compile-Analyzer Compile-Generator Compile-Executor \
             Compile-Blox Link-Objects \
             AST-Test Executor-Test Compile-Test \
             Run-Menhir-Test Blox.tar.gz compiler Demo

# name of blox binary file
EXEC = blox

#name of HelloWorld testfile
HELLO = HelloWorld

# name of test shell script
```

```
TESTSH = testAll

# lexer (scanner) and parser generators
# -v : verbose output option for ocamlyacc
LEXGEN = ocamllex
PARSGEN = ocamlyacc -v

# ocaml compiler and flags
# -c : compile without producing executable files
# -o : specify name of output file produced by compiler
OCC1 = ocamlc -c
OCC2 = ocamlc -o

# source files, generated files, testing, and AMF/output directories
SRC = src
GEN = gen
OBJ = obj
TST = tests
AMF = amf
TARFILES = $(SRC) $(GEN) $(OBJ) $(TST) $(TESTSH).sh README.md Makefile \
                $(TESTFILES:%=tests/%)

# add src, gen, and obj directories to Make path when looking for files
VPATH = src:gen:obj
testfiles := $(wildcard $(TST)/*)

NO_COLOR    = \033[0m
OK_COLOR    = \033[32;01m
OK_STR      = $(OK_COLOR)[OK]$(NO_COLOR)
SUC_STR     = $(OK_COLOR)[BUILD-SUCCESSFUL]$(NO_COLOR)
AWK_CMD     = awk '{ printf "\n%-50s %-10s\n",$$1, $$2; }'
PRINT_OK    = printf "$@ $(OK_STR)"  | $(AWK_CMD)
BUILD_OK    = printf "$@ $(SUC_STR)" | $(AWK_CMD)

compiler:     Clean Create-Scanner Create-AST Create-Parser \
              Compile-Scanner Compile-Parser Compile-Analyzer Compile-Executor \
              Compile-Generator Compile-Blox Link-Objects

Create-Scanner:
	@$(LEXGEN) $(SRC)/scanner.mll
	@mv $(SRC)/scanner.ml $(GEN)/scanner.ml
	@$(PRINT_OK)

Create-Parser:
	@$(PARSGEN) $(SRC)/parser.mly
	@mv $(SRC)/parser.ml $(GEN)/parser.ml
	@mv $(SRC)/parser.mli $(GEN)/parser.mli
	@mv $(SRC)/parser.output $(GEN)/parser.output
```

```
	@#cat $(GEN)/parser.output
	@$(PRINT_OK)

Create-AST:
	@$(OCC1) $(SRC)/ast.ml
	@mv $(SRC)/ast.cmi $(GEN)/ast.cmi
	@mv $(SRC)/ast.cmo $(OBJ)/ast.cmo
	@$(PRINT_OK)

Compile-Scanner:
	@$(OCC1) -I $(GEN) $(GEN)/parser.mli $(GEN)/scanner.ml
	@mv $(GEN)/scanner.cmo $(OBJ)/scanner.cmo
	@$(PRINT_OK)

Compile-Parser:
	@$(OCC1) -I $(GEN) $(GEN)/parser.ml
	@mv $(GEN)/parser.cmo $(OBJ)/parser.cmo
	@$(PRINT_OK)

Compile-Analyzer:
	@$(OCC1) -I $(GEN) $(SRC)/analyzer.ml
	@mv $(SRC)/analyzer.cmo $(OBJ)/analyzer.cmo
	@mv $(SRC)/analyzer.cmi $(GEN)/analyzer.cmi
	@$(PRINT_OK)

Compile-Executor:
	@$(OCC1) -I $(GEN) $(SRC)/executor.ml
	@mv $(SRC)/executor.cmo $(OBJ)/executor.cmo
	@mv $(SRC)/executor.cmi $(GEN)/executor.cmi
	@$(PRINT_OK)

Compile-Generator:
	@$(OCC1) -I $(GEN) $(SRC)/generator.ml
	@mv $(SRC)/generator.cmo $(OBJ)/generator.cmo
	@mv $(SRC)/generator.cmi $(GEN)/generator.cmi
	@$(PRINT_OK)

Compile-Blox:
	@$(OCC1) -I $(GEN) $(SRC)/blox.ml
	@mv $(SRC)/blox.cmo $(OBJ)/blox.cmo
	@mv $(SRC)/blox.cmi $(GEN)/blox.cmi
	@$(PRINT_OK)

Link-Objects:
	@$(OCC2) $(EXEC) $(OBJ)/parser.cmo $(OBJ)/scanner.cmo $(OBJ)/ast.cmo \
	$(OBJ)/analyzer.cmo $(OBJ)/executor.cmo $(OBJ)/generator.cmo $(OBJ)/blox.cmo
	@$(BUILD_OK)
	@echo "\n-------------------------------------------------------\n"
```

```
AST-Test: compiler
	@echo "[$(HELLO).blox:]\n"
	@./$(EXEC) -a $(SRC)/$(HELLO).blox
	@$(PRINT_OK)

Executor-Test: compiler
	@echo "[$(HELLO).blox:]\n"
	@./$(EXEC) -e $(SRC)/$(HELLO).blox
	@$(PRINT_OK)

Compile-Test: compiler
	@echo "[$(HELLO).blox:]\n"
	@./$(EXEC) -c $(SRC)/$(HELLO).blox
	@$(PRINT_OK)

Run-Test-Script:
	@./$(TESTSH).sh
	@mv $(TESTSH).log $(GEN)/$(TESTSH).log
	@sleep .3
	@echo "[Opening $(TESTSH).sh log ...]"
	@cat $(GEN)/$(TESTSH).log
	@$(PRINT_OK)

Run-Menhir-Test:
	menhir --interpret --interpret-show-cst $(SRC)/parser.mly
	@$(PRINT_OK)

Demo: Clean
	@ocamlc -o demo src/exedemo.ml
	@mv src/exedemo.cmi gen/
	@mv src/exedemo.cmo obj/
	@./demo
	@$(PRINT_OK)

Clean:
	@echo "\n-------------------------------------------------------\n"
	@rm -rf $(GEN)/*
	@rm -rf $(OBJ)/*
	@rm -rf $(EXEC)
	@rm -rf demo *.amf
	@$(PRINT_OK)
	@echo "\n"

Blox.tar.gz : $(TARFILES) Clean
	@cd .. && tar czf Blox/Blox.tar.gz $(TARFILES:%=Blox/%)
	@$(PRINT_OK)
```

testAll.sh

```
#!/bin/sh

# Testing script for Blox, adapted from microc testing suite
# Step through a list of files
# Compile, run, and check the output of each expected-to-work test
# Compile and check the error of each expected-to-fail test

# Set time limit for all operations
ulimit -t 30

globallog=testAll.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testAll.sh [options] [.blox files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
       echo "FAILED"
       error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
          SignalError "$1 $2 differs"
          echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
```

```
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
          SignalError "$1 failed on $*"
          return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
        SignalError "failed: $* did not report an error"
        return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                         s/.blox//'`
    reffile=`echo $1 | sed 's/.blox$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
       if [ $keep -eq 0 ] ; then
           rm -f $generatedfiles
       fi
       echo "OK"
       echo "###### SUCCESS" 1>&2
    else
       echo "###### FAILED" 1>&2
       globalerror=$error
    fi
}
```

```
CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.mc//'`
    reffile=`echo $1 | sed 's/.mc$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

# Again not sure what to do here
#    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
#    RunFail "$MICROC" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
#    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
        if [ $keep -eq 0 ] ; then
            rm -f $generatedfiles
        fi
        echo "OK"
        echo "###### SUCCESS" 1>&2
    else
        echo "###### FAILED" 1>&2
        globalerror=$error
    fi
}


while getopts kdpsh c; do
    case $c in
       k) # Keep intermediate files
           keep=1
           ;;
       h) # Help
           Usage
           ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
```

```
then
    files=$@
else
    files="tests/test-*.blox tests/fail-*.blox"
fi

for file in $files
do
    case $file in
       *test-*)
           Check $file 2>> $globallog
           ;;
       *fail-*)
           CheckFail $file 2>> $globallog
           ;;
       *)
           echo "unknown file type $file"
           globalerror=1
           ;;
    esac
done

exit $globalerror
```