BioSyn - A High Level Language For Molecular Synthesis

Introduction

While genetic engineering primarily involves the design, modification and synthesis of individual genes, synthetic biology is an emerging discipline, centered on constructing entire systems of genes and gene products. It involves designing and building new biological systems by assembling molecular and genetic parts, with the purpose of adding to or modifying the biological functions of existing organisms or creating new organisms with specific properties.

In a manner similar to the way electrical engineers assemble complex integrated circuits from transistors, synthetic biology envisions assembly of complex biological devices and networks by assembling individual biological parts. And in a manner similar to how electrical engineers use high level languages such as Verilog and VHDL, Biosyn can be used as a high level language for the purpose of designing biological and genetic assemblies.

Standard biology parts have been catalogued by various parts registries, notable among them being the MIT parts registry (see references below). Assembly standards have been documented by the BioBricks Foundation and made available as RFCs. All parts that are part of an assembly need to comply with one of the assembly standards. Parts compliance data is available in the parts registries. The genetic sequence associated with a part is also published by the parts registries.

Using this language, the biological assembly is put together using high level programming constructs.

Language Tutorial

The language constructs needed to construct an assembly from biological parts are informally described below, together with example usage of the same. In addition to the ability to build assemblies from standard parts, the language includes built in functions to print the genetic sequence, print schematic diagrams, as well as generate XML markup for use with external simulation software.

Primitive types: int, float, char, boolean and string as in C, with the exception that character strings are included as built-in types.

Arrays: arrays of the primitive types as in C

Operators :

arithmetic :	+, -, *, /	
string :	+	// string concatenation as in Java
boolean :	&&, || and !	
Comparison :	>, <, ==, >=, <=, !=	
assignment :	=	// assignment to variables representing primitives, objects and object properties

Conditional and looping constructs – if-else, for and while, as detailed in the language reference manual. These are used in the same way as similar constructs in C.

Complex types: These include types representing biological parts, attributes of parts eg. part sequence, as well as composites derived from parts (assemblies). Prototypes for each of these are defined prior to instantiation as illustrated in the examples below.

```
Attribute Sequence string;          // The sequence property is declared as a string attribute
Attribute Name string;              // The BioBricks name property
Attribute Compatibility string;     // Represents BioBricks RFC compatibility
Attribute Strength float;

// define the prototype for promoters
Part Promoter(Name, Sequence, Compatibility, Strength);

// define the prototype for an RBS
Part RBS(Name, Sequence, Compatibility);

// instantiate the BioBricks promoter BBa_I14018
Promoter Bba_I14018("BBa_I14018", "tgtaagtttatacataggcgagtactctgttatgg", "RFC11", 0.5);

// instantiate the BioBricks RBS Bba_J63003
RBS Bba_J63003("BBa_J63003", "cccgccgccaccatggag", "RFC21");

// create an assembly represented by a composite; composites may contain other composites
Composite Assembly1( Bba_I14018,  Bba_J63003);
```

Constraints: Composites can be parsed and validated against constraints

```
// examples of constraints
Start → <PlasmidBackbone><Prefix><Cassette><Suffix>
Cassette → <Promoter><Cistron><Terminator>
Cistron → <RBS><Gene>
```

User defined functions: Global functions as in C

Library functions: Functions to validate assemblies, print gene sequences, print diagrams and generate markup

```
// examples of function usage
if (Assembly1.validate()) {
        Assembly1.printSequence();
        Assembly1.printDiagram();
        Assembly1.generateMarkup("/usr/local/home/user1/assembly1markup.xml");
}
```

Sample programs listed later illustrate the use of the language both for constructing assemblies, as well as for implementing general purpose iterative algorithms.

Compilation and execution:

Compile the scanner: ocamllex scanner.mll

Compile the parser: ocamlyacc parser.mly

Compile the translator: ocamlc compiler.ml

Compile the driver: ocamlc biosyn.ml

Link all the object files to produce the executable biosyn

Run the compile: biosyn < program.source to produce program.c

Compile program.c to produce the final executable:

```
gcc -o program program.c builtins.c  `pkg-config --cflags --libs gtk+-2.0`
```

Language Reference Manual

1 Program source

1.1 Whitespace

Whitespace characters such as spaces and tabs are used to separate tokens in the input and are discarded during parsing.

1.2 Comments

Single line and multi-line comments are supported. Single line comments are denoted by the characters // and continue to the end of the line. These may be placed on the same line as the source code.

Multi-line comments begin with a /* and end with a */. They may not be embedded within each other.

1.2 Semicolons

Semicolons are used to terminate statements. Multiple statements may be placed on the same line by placing semicolons between them. Multiple consecutive semicolons are considered as a single statement terminator.

1.4 Identifiers

All identifiers begin with a letter and may be followed by zero or more letters, digits or underscores. Identifiers are case sensitive.

1.5 Literals

Literals in the language can be integers, characters, strings, booleans or floating point numbers.

1.6 Keywords

The following identifiers are reserved as keywords in the language:

char	int	float	string	boolean
for	while	if	else	break
continue	true	false	return	function
void	Attribute	Part	Composite	

1.7 Other tokens

The following characters have meaning in the language:

```
+    -    /    *    =    >    <    >=   <=   ==

!    !=   ||   &&   {    }    (    )    [    ]

,    .    [|   :    ->
```

2 Fundamental types

2.1 Integers

An integer consists of one or more consecutive digits. Integers are assumed to be base 10 and may not begin with a zero.

2.2 Floats

These are arbitrary precision decimals and are composed of three parts, all of which are optional:

- An integer part composed of digits
- A decimal part consisting of a period followed by digits
- An exponent part consisting of an e or E, followed by an optional + or -, followed by digits

Either the integer or the decimal part must be present.

2.3 Characters

Characters consist of a single character enclosed in single quotes

2.4 Strings

These are sequences of characters. String literals are sequences of characters enclosed in double quotes. The double quote character is escaped with a \

2.5 Booleans

The boolean constants true and false are language keywords and represent logical truth and falsehood

2.6 Attributes

Attributes are labels for primitives associated with parts. Attribute definitions must specify a primitive type.
Examples of attribute definitions are:

```
Attribute Sequence string;
Attribute Strength float;
```

3 Composite types

3.1 Arrays

These are one dimensional fixed length zero indexed arrays. Arrays are homogeneous ie. they may consist of objects of a single type. Array literals may be declared by enclosing a list of objects in square brackets and separating the elements by commas.

3.2 Parts

Part definitions define prototypes for standard biological parts. Parts can be instantiated only after the prototypes are defined. Parts have attributes which must be defined before parts can be defined.

```
Attribute Sequence string;
Attribute Name string ;
Attribute Compatibility string;
Attribute Strength float;

// define the prototype for promoters
Part Promoter(Name, Sequence, Compatibility, Strength);
```

3.3 Composites

Composites represent composites of biological parts or other composites. The same part or composite can be reused in a composite declaration.

```
// instantiate the BioBricks promoter BBa_I14018 based on the prototype
// declared above
Promoter Bba_I14018("BBa_I14018", "tgtaagtttatacataggcgagtactctgttatgg",
"RFC21", 0.5);
// instantiate the BioBricks RBS Bba_J63003
RBS Bba_J63003("BBa_J63003", "cccgccgccaccatggag", "RFC21");
// instantiate the BioBricks terminator BBa_B1002
Terminator Bba_B1002("BBa_B1002", "cgcaaaaaaccccgcttcggcggggtttttttcgc",
"RFC21");

// create an assembly, represented by a composite; composites may contain other
//composites
Composite Assembly1( Bba_I14018, Bba_J63003, Bba_B1002);
```

4 Expressions

4.1 Arithmetic expressions

Arithmetic expressions consist of binary operators, the unary negation operator and parenthesis. Parentheses have the highest precedence, followed by the unary negation operator, followed by * and /, and then + and -. Arithmetic operations may be performed on integers as well as floats.

4.2 String concatenation

The + operator is overloaded for performing string concatenation.

4.3 Relational expressions

These consist of expressions involving >, <, >=, ==, != and <=. These operations may be performed on integers or floats, however operations on floats may not yield expected results.

4.4 Logical expressions

Logical expressions involve the !, || or && operators. These are performed on boolean operands and return the boolean values of true or false.

4.5 Attribute access expressions

Part attributes may be accessed using the colon operator as in:

if (promoter1:RelativeStrengh > 5.0) ….

4.6 Composite part access expressions

Parts within a composite may be accessed using the square bracket notation.
Parts within a composite are zero indexed. Using the declarations in 3.3,

Promoter Bba_I14018 = Assembly1[0];

4.7 Array access expressions

Array are zero indexed, and their elements can be accessed using the square bracket notation.

4.8 Function calls

Functions are invoked by specifying the name of the function followed by a comma-separated list of parameters contained within "[|" and "|]". The parameter list is optional but the delimiters are not. Functions may have return types, in which case, the expression has the same type as the function.

4.9 Identifiers and literals

As specified in 1.4 and 1.5

5 Statements

Statements could be expressions or one or more of the statement types described below and terminated by a semicolon. If the statement consists of an expression only, its value is discarded. Statements may be grouped in blocks. Program flow proceeds from top to bottom, unless a conditional or iterative statement is encountered.

5.1 Statement blocks

Statements may be grouped between { and }. Each statement within the block must be terminated with a semicolon.

5.2 Prototype definitions

As defined in 3.2

5.3 Declarations

All variables must be declared before they can be used. Declaration may be accomplished in conjunction with assignment. The declaration syntax is:

<*type*> <*identifier*>;

To declare an array, use:

<*type*> <*identifier*>[<*size*>];

For declaration with assignment:

<*type*> <*identifier*> = <*expression*>;

examples:

```
int myVar;
int myVar2 = 10;
int myArray[5];

int myArray2[5] = [1,2,3,4,5];
```

5.4 Assignment

The assignment operator = is used to assign an expression to an identifier.

Identifiers must be declared before they can be assigned. Declaration and assignment may be performed in the same statement.

For the variable declared in 5.3, we have:

myVar = 10;

5.5 Conditional statements

The if statement may have an optional else part. The two forms of the if statement are:

if <*boolean expression*> <*statement*>
if <*boolean expression*> <*statement*> else <*statement*>

If-else semantics follow C semantics. <*statement*> may include multiple statements within { and }.

5.6 Iterative statements

These have two forms:

while (<*boolean expression*>) <*statement*>

for *(<expression-1>; <expression-2>; <expression-3>)* <*statement*>

These follow C semantics.

5.7 Function definitions

These follow the syntax:

```
<type> function_name <variable list> {
            <statement>
}
```

Functions may not be nested or recursive. The return type may also be an array or a user defined type. Functions may return void.

5.8 Return

The return statement occurs within function definitions and may return void or an expression.

5.9 Break

The break statement occurs in loops and returns control to the statement following the loop.

5.10 Continue

The continue statement occurs within loops and causes program flow to begin the next iteration.

5.11 Constraints

Constraints define rules for constructing assemblies. Composites are parsed and validated against constraints.

```
// examples of constraints
Start → <PlasmidBackbone><Prefix><Cassette><Suffix>
Cassette → <Promoter><Cistron><Terminator>
Cistron → <RBS><Gene>
```

In this manner the programmer is constrained to build only meaningful assemblies.

6 Predefined functions

6.1 validate()

The validate function validates composite assemblies against the constraints that have been defined.

6.2 printSequence()

Prints the genetic sequence of a part or an assembly of parts.

6.3 printDiagram()

Prints a schematic diagram of an assembly by associating an icon with each part.

6.4 generateMarkup()

Generates XML markup representing the assembly, which can be input to external simulation software.

Example usage of these function is as follows:

```
// validate the assembly against the declared constraints and then print sequence, print
// diagram and generate markup
if (Assembly1.validate()) {
    Assembly1.printSequence();
    Assembly1.printDiagram();
    Assembly1.generateMarkup("/usr/local/home/user1/assembly1markup.xml");

}
```

Project Plan

Planning:

Stages involved in planning were:
Initial research and development of the initial proposal
Refinement of the initial proposal
Further research into the problem domain
Identifying feature set and development of the language reference manual
Preliminary development of lexer and parser code
Resolution of ambiguities
Development of compile time type checking and translation code
Testing
Documentation

Coding Style:

The Ocaml programming guidelines, available at
http://caml.inria.fr/resources/doc/guides/guidelines.en.html were followed for this project.

Project Timeline:

March - planning, research, language reference manual

April – scanner and parser, resolution of ambiguities

May – development of the compiler, testing and debugging, working on implementation details

Development Environment:

Ocamllex, Ocamlyacc, Ocaml compiler, Aquamacs with tuareg mode, GTK+ ver 2

Architecture

Scanner ->Parser -> Ast -> Static semantic analysis, translation to C-> Compilation, linking C->Execution

The scanner, scanner.mll was developed for Ocamllex.
The parser, parser.mly was developed for Ocmlyacc.
The AST types are defined in ast.ml.
Type checking, compile time semantic analysis and translation are in compiler.ml
The resulting C code is linked with builtins.c, which contains code for predefined functions, as well as with GTK libraries to produce the final executable.

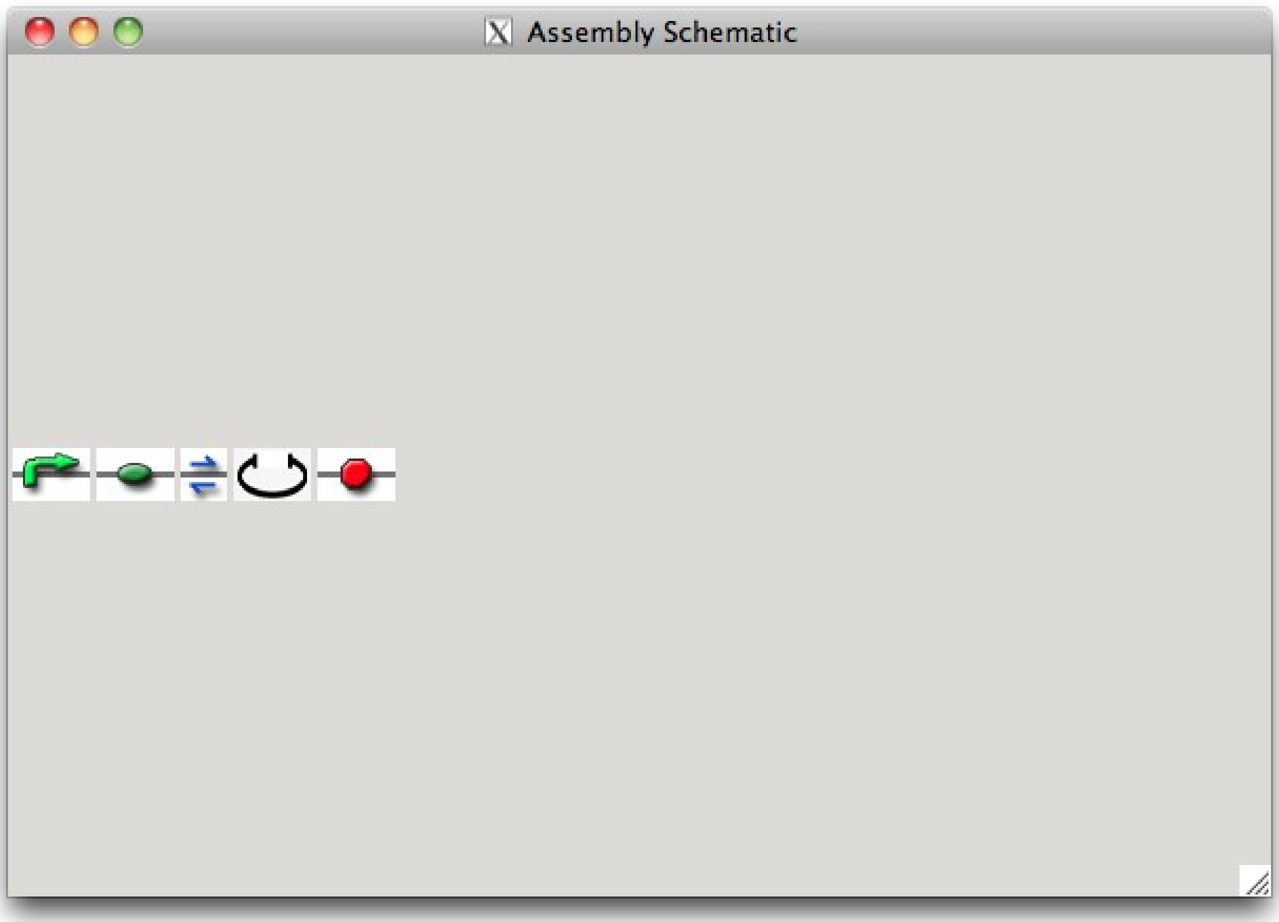
Test Plan

A number of test cases were developed, however the tests were to be carried out manually. Given the various output formats generated by the programs, it is difficult to completely automate the testing process.

Lessons Learnt

A correct estimation of the scope and complexity of the as well as the learning curve associated with tools such as lex and yacc as well as the Ocaml language itself, would have been helpful. The project turned out to be larger and more complex than initially estimated.

Sample Output



```
<?xml version=\"1.0\" encoding=\"UTF-8\" ?>
<Parts>
          <Part>
               <Name>
                 BB_Promoter1
               </Name>
              <Prototype>
                     Promoter
              </Prototype>
              <Attributes>
                      <Attribute>
                              <Name>
                                Identifier
                              </Name>
                              <Value>
                                001
                              </Value>
                       </Attribute>
                       <Attribute>
                              <Name>
                                Sequence
                              </Name>
                              <Value>
                                gcaaccattatcaccgccagaggtaaaatagtcaacacgcacggtgtta
                              </Value>
                       </Attribute>
             </Attributes>
          </Part>
          <Part>
               <Name>
                 BB_RBS1
               </Name>
              <Prototype>
                RBS
              </Prototype>
              <Attributes>
                      <Attribute>
                              <Name>
                                Identifier
                              </Name>
                              <Value>
                                001
                              </Value>
                       </Attribute>
                       <Attribute>
                              <Name>
                                Sequence
                              </Name>
                              <Value>
```

```
                    attaaagaggagaaa
                  </Value>
              </Attribute>
        </Attributes>
      </Part>
</Parts>
```

Sequence is:
cgatgtacgggccagatatacgcgttgacattgattattgcctagttattaatagtaatcaattacggggtcattagttcatagcccatat
atggagttc
cgcgttacataacttacggtaaatggcccgcctggctgaccgcccaacgacccccgcccattgacgtcaataatgacgtatgttcc
catagtaacgccaa
tagggactttccattgacgtcaatgggtggagtatttacggtaaactgcccacttggcagtacatcaagtgtatcatatgccaagtacg
ccccctattga
cgtcaatgacggtaaatggcccgcctggcattatgcccagtacatgaccttatgggactttcctacttggcagtacatctacgtattagt
catcgctatt
accatggtgatgcggttttggcagtacatcaatgggcgtggatagcggtttgactcacggggatttccaagtctccaccccattgacg
tcaatgggagtt
tgttttggcaccaaaatcaacgggactttccaaaatgtcgtaacaactccgccccattgacgcaaatgggcggtaggcgtgtacgg
tgggaggtctatat
aagcagagctctctggctaactagagaacccactgcttactggcttatcgaaattactagatggagcagaagctgatcagcgagg
aggactactagatgg
attatcaagtgtcaagtccaatctatgacatcaattattatacatcggagccctgccaaaaaatcaatgtgaagcaaatcgcagccc
gcctcctgcctcc
gctctactcactggtgttcatctttggttttgtgggcaacatgctggtcatcctcatcctgataaactgcaaaaggctgaagagcatgac
tgacatctac
ctgctcaacctggccatctctgacctgtttttccttcttactgtccccttctgggctcactatgctgccgcccagtgggactttggaaatac
aatgtgtc
aactcttgacagggctctattttataggcttcttctctggaatcttcttcatcatcctcctgacaatcgataggtacctggctgtcgtccatg
ctgtgtt
tgctttaaaagccaggacggtcacctttggggtggtgacaagtgtgatcacttgggtggtggctgtgtttgcgtctctcccaggaatcat
ctttaccaga
tctcaaaaagaaggtcttcattacacttgcagctctcattttccatacagtcagtatcaattctggaagaatttccagacattaaagata
gtcatcttgg
ggctggtcctgccgctgcttgtcatggtcatctgctactcgggaatcctaaaaactctgcttcggtgtcgaaatgagaagaagaggc
acagggctgtgag
gcttatcttcaccatcatgattgtttattttctcttctgggctccctacaacattgtccttctcctgaacaccttccaggagttctttggcctgaa
taat
tgcagtagctctaacaggttggaccaagctatgcaggtgacagagactcttgggatgacgcactgctgcatcaaccccatcatctat
gcctttgtcgggg
agaagttcagaaactacctcttagtcttcttccaaaagcacattgccaaacgcttctgcaaatgctgttctattttccagcaagaggctc
ccgagcgagc
aagctcagtttacacccgatccactggggagcaggaaatatctgtgggcttgatgcagattttcgtcaagactttgaccggtaaaac
cggaacattggaa
gttgaatcttccgataccatcgacaacgttaagtcgaaaattcaagacaaggaaggaatccctggtggatgtcgacctcgagtcat
gtaa

Source Listing

scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }          (* Comments *)
| "//"[^ '\n']*     { token lexbuf }       (* Single line comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRACKET }
| ']'      { RBARCKET }
| ';'      { SEMI }
| ','      { COMMA }
| ':'      { COLON }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "|"      { DELIM }
| '!'      { NOT }
| '.'      { DOT }
| "->"     { CONSTRAINEDBY }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "break"  { BREAK }
| "continue" { CONTINUE }
| "int"    { INT }
| "float"  { FLOAT }
| "string" { STRING }
| "char"   { CHAR }
| "bool"   { BOOL }
| "void"   { VOID }
| "Attribute" { ATTRIBUTE }
| "Part"      { PART }
| "Composite" { COMPOSITE }
| ['0'-'9']+ as lxm { INTLITERAL(int_of_string lxm) }
| '\"'(['\\"\"']|[^ '\n'  '\"'])*'\"' as lmx { STRINGLITERAL(lxm) }
| ['\"][^'\n']['\"] as lmx { CHARLITERAL(lxm) }
| ['-']?((['0'-'9']+)|(['0'-'9']*'.'['0'-'9']+)(['e"E']['+"-']?['0'-'9']+)?) as lxm { FLOATLITERAL(float_of
string lxm) }
| "true" as lxm { BOOLEANLITERAL(lxm) }
| "false" as lxm { BOOLEANLITERAL(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| "Promoter" as lxm { TYPE(lxm) }
| "Prefix" as lxm { TYPE(lxm) }
| "Suffix" as lxm { TYPE(lxm) }
| "RBS" as lxm { TYPE(lxm) }
| "Terminator" as lxm { TYPE(lxm) }
| "Gene" as lxm { TYPE(lxm) }
| "Cistron" as lxm { TYPE(lxm) }
| "Cassette" as lxm { TYPE(lxm) }
| "Plasmid" as lxm { TYPE(lxm) }
| "PlasmidBackbone" as lxm { TYPE(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  ['*"/'] { token lexbuf }
| _    { comment lexbuf }
```

parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA
COLON
%token PLUS MINUS TIMES DIVIDE ASSIGN DOT
%token AND OR NOT
%token EQ NEQ LT LEQ GT GEQ
%token RETURN BREAK CONTINUE IF ELSE FOR WHILE INT CHAR STRING BOOL
FLOAT ATTRIBUTE PART COMPOSITE CONSTRAINEDBY TYPE VOID DELIM
%token <int> INTLITERAL
%token <char> CHARLITERAL
%token <string> STRINGLITERAL
%token <bool> BOOLEANLITERAL
```

```
%token <float> FLOATLITERAL
%token <string> ID
%token <string> TYPE
%token EOF

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

%start program
%type <Ast.program> program

%%

program:
  /* nothing */ { [], [] }
 | program stmt { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
  INT ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
    { { returntype = StringLiteral("int");
       fname = $2;
         formals = $4;
         body = List.rev $7 } }
  | CHAR ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
    { { returntype = StringLiteral("char");
       fname = $2;
         formals = $4;
         body = List.rev $7 } }
  | FLOAT ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
    { { returntype = StringLiteral("float");
       fname = $2;
         formals = $4;
         body = List.rev $7 } }

  |BOOL ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
    { { returntype = StringLiteral("bool");
       fname = $2;
         formals = $4;
         body = List.rev $7 } }
  | STRING ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
    { { returntype = StringLiteral("string");
```

```
        fname = $2;
          formals = $4;
          body = List.rev $7 } }
  | VOID ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
     { { returntype = StringLiteral("void");
        fname = $2;
          formals = $4;
          body = List.rev $7 } }
 | TYPE ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
     { { returntype = StringLiteral($1);
        fname = $2;
          formals = $4;
          body = List.rev $7 } }

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

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

attribute_list:
    ID                  { [$1] }
  | attribute_list COMMA ID { $3 :: $1 }

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

stmt:
  expr SEMI { Expr($1) }
  | INT ID SEMI { PrimitiveDeclaration(Int, $2, None) }
  | CHAR ID SEMI { PrimitiveDeclaration(Char, $2, None) }
  | FLOAT ID SEMI { PrimitiveDeclaration(Float, $2, None) }
  | BOOL ID SEMI { PrimitiveDeclaration(Bool, $2, None) }
  | STRING ID SEMI { PrimitiveDeclaration(String, $2, None) }
  | INT ID ASSIGN expr { PrimitiveDeclaration(Int, $2, Some($4)) }
  | CHAR ID ASSIGN expr { PrimitiveDeclaration(Char, $2, Some($4)) }
  | FLOAT ID ASSIGN expr { PrimitiveDeclaration(Float, $2, Some($4)) }
  | BOOL ID ASSIGN expr { PrimitiveDeclaration(Bool, $2, Some($4)) }
  | STRING ID ASSIGN expr { PrimitiveDeclaration(String, $2, Some($4)) }
  | expr ID LBRACKET expr RBRACKET ASSIGN actuals_list SEMI { ArrayDeclaration($1, $2,
$4, Some(List.rev $7)) }
  | expr ID LBRACKET expr RBRACKET SEMI { ArrayDeclaration($1, $2, $4, None) }
  | ATTRIBUTE ID primitive_type SEMI { Attribute($2, $3) }
  | PART TYPE LPAREN attribute_list RPAREN SEMI { Prototype ($2, List.rev $4) }
  | ID CONSTRAINEDBY rhs SEMI { Constraint($1, $3) }
  | BREAK SEMI { Break }
  | CONTINUE SEMI { Continue }
  | RETURN expr SEMI { Return($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) }

rhs:
    LT ID GT { [$2] }
  | rhs LT ID GT { List.rev $3 :: $1 }

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

expr:
  | INTLITERAL          { IntLiteral($1) }
  | BOOLEANLITERAL          { BooleanLiteral($1) }
  | CHARLITERAL          { CharLiteral($1) }
  | FLOATLITERAL          { FloatLiteral($1) }
  | STRINGLITERAL          { StringLiteral($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 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) }
  | NOT expr         { Negation($2) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LBRACKET INTLITERAL RBRACKET ASSIGN expr { ArrayUpdate($1, $3, $6) }
  | TYPE  ID  LPAREN actuals_list RPAREN { Part($1, $2, List.rev $4) }
  | COMPOSITE ID LPAREN attribute_list RPAREN { Composite($2, List.rev $4) }
  | ID COLON ID { PartAttribute($1, $3) }
   | ID LBRACKET INTLITERAL RBRACKET { CollectionMember($1, $3) }
  | ID  LBRACKET DELIM actuals_opt DELIM RBRACKET  { Call($1, $4) }
  | ID DOT ID LPAREN RPAREN { BuiltInCall($3, $1) }
  | LPAREN expr RPAREN { $2 }
  | ID       { Id($1) }

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

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

primitive_type:
    INT    { Int }
  | CHAR   { Char }
  | BOOL   { Bool }
  | STRING { String }
  | FLOAT  { Float }
```

ast.ml

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | And | Or | Not

(* type boolop =  Equal | Neq | Less | Leq | Greater | Geq | And | Or *)

type p_type = Pint of int| Pchar of char | Pbool of bool | Pfloat of float | Pstring of string

type primitive_type = Int | Char | Bool | Float | String


type env2 = {
    primitives : (string, string) Hashtbl.t ;
    attributes : (string, string) Hashtbl.t;
    arrays : (string, string * int) Hashtbl.t;
    prototypes : (string, string list) Hashtbl.t;
    parts : (string, string * string list) Hashtbl.t;
    composites : (string, string list) Hashtbl.t;
    functions : (string, string)Hashtbl.t;
    builtins: string list;
    constraints: (string, string list)Hashtbl.t;

  }

type expr =
    IntLiteral of int
  | BooleanLiteral of bool
  | CharLiteral of char
  | FloatLiteral of float
  | StringLiteral of string

  | Id of string
  | Binop of expr * op * expr
  | Negation of expr
  | Assign of string * expr
  | ArrayUpdate of string * int * expr (* Array name, index, value *)
  | Part of string * string * expr list (* Prototype name, part name, attribute values - part
instantiation *)
  | Composite of string * string list (* Name,  list of parts or composites - composite
instantiation *)
  | PartAttribute of string * string (* Part name, attribute name  - attribute access*)
  | CollectionMember of string * int (* Composite name, index  - composite member access*)
  | Call of string * expr list
  | BuiltInCall of string * string
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | PrimitiveDeclaration of primitive_type * string * expr option (* Type, name, optional value *)
  | ArrayDeclaration of string * string * expr * expr list option (* Type, name, size, optional
values *)
  | Attribute of string * primitive_type (* Attribute name, type *)
  | Prototype of string * string list (* Name, list of attribute names - part prototype definition *)
  | Constraint of string * string list (* Head, list of RHS symbols - constraint definition *)
  | Break
  | Continue
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

type func_decl = {
    returntype : string;
    fname : string;
    formals : string list;
    locals : string list;
    body : stmt list;
  }

type program = stmt list * func_decl list

let rec string_of_expr = function
  IntLiteral(i) -> string_of_int i
  | StringLiteral(s) -> s
  | CharLiteral(c) -> "'" ^ Char.escaped c ^ "'"
  | FloatLiteral(f) -> string_of_float f
  | BooleanLiteral(b) -> string_of_bool b
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
          Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!=" | And -> "&&" | Or -> "||" | Not -> "!"
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=") ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Negation(e) ->  "!" ^ string_of_expr e
  | Call(f, el) ->
      f ^ "[|" ^ String.concat ", " (List.map string_of_expr el) ^ "|]"
  | Noexpr -> ""
  | ArrayUpdate(s, i, e) -> s ^ "[" ^ string_of_int i ^ "] = " ^ string_of_expr e
  | Part(s1, s2, elist) -> s1 ^ s2 ^ "(" ^ String.concat ", " (List.map string_of_expr elist) ^ ")"
  | Composite(s, l) -> "Composite " ^ s ^ "(" ^ String.concat ", " (List.map (fun x -> x) l) ^ ")"
  | PartAttribute(s1, s2) -> s1 ^ ":" ^ s2
  | CollectionMember(s, i) -> s ^ "[" ^ string_of_int i ^ "]"
  | BuiltInCall (s1, s2) -> s1 ^ ".(" ^ s2 ^ ")"

let string_of_op op =
  match op with
          Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
      | And -> "&&" | Or -> "||" | Not -> "!"

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\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
  | PrimitiveDeclaration(p, s, v) -> (* Primitive type, name, optional value *)
      ( match p with
          | Int ->  "int "
          | Char -> "char "
          | Float -> "float "
          | Bool ->"bool "
          | String ->"string " )
      ^ s ^
         ( match v with
           | None -> ""
           | Some v1 ->" = " ^  string_of_expr v1 ^ ";" )
  | ArrayDeclaration(s1, s2, i, l) -> s1 ^ " " ^ s2 ^ "[" ^ string_of_expr i ^ "]" ^ (* Type, name,
size, optional list of values *)
```

```
   ( match l with
            | None -> ""
            | Some l1 ->" = {"  ^ String.concat ", " (List.map string_of_expr l1) ^ "};" )
  | Attribute(s, p) -> "Attribute " ^ s ^ " " ^ (* Attribute name, primitive type *)
     ( match p with
           | Int ->  "int;"
           | Char -> "char;"
           | Float -> "float;"
           | Bool ->"bool;"
           | String ->"string;" )
  (* Name, list of attribute names - part prototype definition *)
  | Prototype(s, l) -> "Part " ^ s ^ "(" ^ String.concat ", " (List.map (fun x -> x) l) ^ ");"
  (* Head, list of RHS symbols - constraint definition *)
  | Constraint(s, l) -> s ^ "->" ^ "<" ^ String.concat "><" (List.map (fun x -> x) l) ^ ">;"
  | Break -> "break"
  | Continue -> "continue"

let string_of_vdecl id = "int " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  fdecl.returntype ^ " " ^ fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (stmts, funcs) =
  String.concat "\n" (List.map string_of_stmt stmts) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)



Open Ast

(** Translate a program in AST form into a C program.  Throw an
    exception if something is wrong, e.g., a reference to an unknown
    variable or function *)
let translate (statements, functions) =

(** All global statements are placed in a main function *)
  let main = {
    returntype = "int";
    fname = "main";
    formals = [];
    locals = [];
    body = statements;
  } in

  (** main() is added to list of functions *)
  let allfunctions = main :: functions in
  (** Output file *)
  let file = open_out "program.c" in

  (** Translate a function in AST form into a C program *)
  let process env fdecl =

    (** Get the type of expressions *)
    let rec get_type expression =
      match expression with
          IntLiteral(i) -> "int"
        | StringLiteral(s) -> "string"
        | CharLiteral(c) -> "char"
        | FloatLiteral(f) -> "float"
        | BooleanLiteral(b) -> "bool"
        | Id s -> "string"
        | Binop (e1, op, e2) -> if ((get_type e1) == (get_type e2)) then get_type e1

          else if ( ((get_type e1) == "int" && (get_type e2) == "float") || ((get_type e1) == "float"
&& (get_type e2) == "int")) then "float"
          else "none"
        | Negation(e) -> get_type e
        | Assign (s, e) ->  if ((get_type Id(s)) == (get_type e)) then get_type e else "none"
        | ArrayUpdate(s, i, e) -> if ((fst (Hashtbl.find env.arrays s)) == (get_type e)) then get_type
e else "none"
        | Part(s1, s2, elist) -> s1
        | Composite(s, l) -> "none"
        | PartAttribute(s1, s2) -> Hashtbl.find env.attributes s2
        | CollectionMember(s, i) -> if (Hashtbl.mem env.arrays s) then (fst (Hashtbl.find
env.arrays s)) else "none"
        | Call (fname, actuals) -> "none" (* todo - table of function definitions ?? *)
        | BuiltInCall (s1, s2) -> if (s2 == "validate") then "bool" else "void"
        | Noexpr ->   "none"
    in

    (** Convert heterogeneous types into a single type *)
    let get_types_attribute attr =
      match attr with
          IntLiteral(i) ->  Pint(i)
           | BooleanLiteral(b) -> Pbool(b)
           | CharLiteral(c) -> Pchar(c)
           | FloatLiteral(f) -> Pfloat(f)
           | StringLiteral(s) -> Pstring(s)
    in

    (** Get index of and attribute in a part *)
    let get_index s l =
      let rec iter i n l =
           match l with
        | hd::tl -> iter (i+1) (if hd = s then i else n) tl
        | [] -> n
  in
  iter 0 (-1) l
in

(** Create the XML string for a composite *)
let get_xml_string composite =

  let parts =   Hashtbl.find env.composites composite in
  let prototypes =  List.map (fun x  -> fst ( Hashtbl.find env.parts x) ) parts  in
  let attributes = List.map (fun x  -> Hashtbl.find env.prototypes x)
    prototypes  in

  let actuals = List.map (fun x -> snd( Hashtbl.find env.parts x))
    parts  in

  "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n" ^ "<Parts>\n" ^
      (* map through each part *)
      String.concat ""
      (List.map(fun x ->
        let partindex = get_index x parts in
        let part_prototype = List.nth prototypes partindex in
        let part_attributes = List.nth attributes partindex in
        let part_actuals = List.nth actuals partindex in


        "\t<Part>\n" ^
          "\t\t<Name>\n" ^
          "\t\t\t" ^ x ^ "\n" ^
          "\t\t<Name>\n" ^
          "\t\t<Prototype>\n" ^
          "\t\t\t" ^ part_prototype ^ "\n" ^
          "\t\t<Prototype>\n" ^
          "\t\t<Attributes>\n" ^

          (* attrbutes here *)
          String.concat ""
          (List.map2 (fun x y ->
            "\t\t\t<Attribute>\n" ^
                "\t\t\t\t<Name>\n" ^
                "\t\t\t\t\t" ^ x ^ "\n" ^
                "\t\t\t\t</Name>\n" ^
                "\t\t\t\t<Value>\n" ^
                "\t\t\t\t\t" ^ y ^ "\n" ^
                "\t\t\t\t</Value>\n" ^
           "\t\t\t</Attribute>\n"
           ) part_attributes part_actuals) (* inner fun *) ^
```

```
            "\t\t<Attributes>\n" ^
            "\t<Part>\n"
          )  parts) (* fun *) ^ "<Parts>\n"
   in

   (** Concatenate the sequences for all parts in a composite *)
   let get_sequence_string composite =
     String.concat ""
         ( List.map (fun x ->
           let index = get_index "Sequence" ( Hashtbl.find env.prototypes (fst (Hashtbl.find
env.parts x)) )
           in
          (List.nth (snd ( Hashtbl.find env.parts x)) index )) (Hashtbl.find env.composites
composite) )
   in
     let get_part_sequence_string part =
       let index = get_index "Sequence" (Hashtbl.find env.prototypes (fst ( Hashtbl.find
env.parts part)))
       in
         (List.nth (snd ( Hashtbl.find  env.parts part)))
   in

   (** Translate an expression - see ast.ml for comments *)
   let rec expr e =  match e with
       IntLiteral(i) -> Printf.fprintf file (string_of_int i) ^ " "
     | BooleanLiteral(b) -> Printf.fprintf file (string_of_bool b) ^ " "
     | CharLiteral(c) -> Printf.fprintf file  "'" ^ (string_of_char c) ^ "' "
     | FloatLiteral(f) -> Printf.fprintf file (string_of_float f) ^ " "
     | StringLiteral(s) -> Printf.fprintf s ^ " "
     | Id s ->
         if (Hashtbl.mem env.primitives s )
         then    Printf.fprintf file s ^ " "
       else
         raise (Failure ("undeclared variable " ^ s))
     | Binop (e1, op, e2) ->
         if ( ( (get_type e1) == "int" || (get_type e1) == "float")
                              &&  ( (get_type e2) == "int" || (get_type e2) == "float")
                            && !(op == And || op == Or ))
           then  Printf.fprintf file string_of_expr e1 ^ string_of_op op ^ " " ^ string_of_expr e2
         else if ( ( (get_type e1) == "string" && (get_type e1) == "string") && op == Add)
         then     Printf.fprintf file "strcat(" ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^ ")"
         else
           raise (Failure ("type mismatch in binary operation " ^ string_of_expr (Binop (e1, op,
e2)) ))
     | Negation(e) ->
         if ((get_type e) == "bool")
         then   Printf.fprintf file "!(" ^ expr e ^ ") "
         else
           raise (Failure ("type mismatch in negation operation " ^ string_of_expr e))
```

```
    | Assign (s, e) ->
        (try if ((Hashtbl.find env.primitives s) == (get_type e)) then
         Printf.fprintf file s ^ " = " ^ string_of_expr e ^ ";" else ()
        with Not_found -> ( try if ((Hashtbl.find env.parts s) == (get_type e)) then
          Printf.fprintf file s ^ " = " ^ string_of_expr e ^ ";" else ()
          with Not_found -> raise (Failure ("undeclared variable " ^ s)) ))
    | ArrayUpdate(s, i, e)->
      (* todo - can assign int to float ?? *)
      (try if  (( fst (Hashtbl.find env.arrays s) == (get_type e)) &&
              (snd (Hashtbl.find env.arrays s) < i))
         then   Printf.fprintf file s ^ string_of_int i ^ " = " ^ string_of_expr e ^ ";"
         else
           raise (Failure ("array index out of bounds"))
        with Not_found ->
        raise (Failure ("type mismatch in array operation")) )
    | Part(s1, s2, elist) ->
      (try  let _ =  List.iter2 (fun x y -> if (x != get_type y) then raise (Failure ("type mismatch in
part instantiation")) else ())
             (Hashtbl.find env.prototypes s1) elist in
           let attributes = String.concat ", " (List.map (fun x -> string_of_expr x ) elist) in
           let _ =  Printf.fprintf "struct " ^ s1 ^ " " ^ s2 ^ " = {" ^ attributes  ^ "};"  in
           Hashtbl.add env.parts s2 (s1, elist)
        with Not_found ->
        raise (Failure ("type mismatch - undefined part prototype")) )
    | Composite(s, l) ->
        if (Hashtbl.mem env.composites s)
        then raise (Failure ("composite has already been defined"))
        else let l1 = List.concat(List.map (fun x -> if (Hashtbl.mem env.composites x)
                                         then Hashtbl.find env.composites
                                         else x) l)
        in
             let _ =   Hashtbl.add env.composites s l1 in
            Printf.fprintf file "Composite " ^ s ^ "(" ^
            String.concat "," List.map (fun x ->  x ) l1 ^ ");\n"
    | PartAttribute(s1, s2) ->
      (try  let prototype =  Hashtbl.find env.parts s1 in
           let attributes = Hshtbl.find env.prototypes prototype in
           List.map (fun x y -> if ( x != y) then raise (Failure ("part " ^ s1 ^ ", attribute " ^ s2 ^ "
not found"))
                               else  Printf.fprintf file s1 ^ "." ^ s2  attributes) s2
        with Not_found ->
        raise (Failure ("undefined part or attribute")))
    | CollectionMember(s, i) ->
      if ((Hashtbl.mem env.primitives s)) then
        Printf.fprintf file s ^ "[" ^ string_of_int i ^ "];"
         else if ((Hashtbl.mem env.composites s)) then
        Printf.fprintf file s ^ "[" ^ string_of_int i ^ "];"
         else  raise (Failure ("undeclared array or composite " ^ s))
    | Call (fname, actuals) ->
```

```
      (* todo argument type checking - can be done in one pass ? *)
        if ((fname == "print") && (List.length actuals > 1))
      then raise (Failure ("print function can have only one argument"))
      else  let actuals_string =  String.concat "," ( List.map (fun x -> string_of_expr x ) actuals)
in
          Printf.fprintf file fname ^ ".(" ^ actuals_string ^ ");"
    | BuiltInCall(s1, s2) ->
      (try let parts = Hashtbl.find env.composites s1 in
          let _ =  Printf.fprintf file "char *composite_" ^ s1 ^ "[] = {\"" ^
              String.concat "\" , \"" List.map (fun x ->(fst Hashtblfind env.parts)) parts ^ "\"};\n" in
            (match s2 with
                 "validate" -> Printf.fprintf file s2 ^ "." ^ s2 ^  "();\n"
            | "printSequence" -> Printf.fprintf file s1 ^ "." ^ s2 ^ "(" ^ get_part_sequence_string
s1 ^ ");\n"
            | "printDiagram" ->  Printf.fprintf file s1 ^ "." ^ s2 ^ "(" ^  "composite_" ^ s1 ^ ");\n"
              | "generateMarkup" -> Printf.fprintf file s1 ^ "." ^ s2 ^ "(\"markup.xml\"," ^
get_xml_string s1 ^ ");\n")
       with Not_found -> if (( List.mem env.builtins s2) != true)
                      then raise (Failure ("unknown method " ^ s2))
                      else
                        (try  let prototype = Hashtbl.find env.parts s1 in
                          if (s2 != "printSequence")
                          then raise (Failure ( "method " ^ s2 ^ " is only valid for composites"))
                          else Printf.fprintf file s ^ "." ^ s2 ^ "(" ^ get_part_sequence_string  s2 ^
");\n"
                       with Not_found -> raise (Failure ("unknown composite or part"))) )
    | Noexpr -> ()
   in

   (** Translate a statement *)
   let rec stmt st = match st with
     Block s -> let _ =  Printf.fprintf file "{\n"  in let _ =  List.iter (fun x -> stmt x)  sl in
Printf.fprintf file "}\n"
    | Expr e       -> let _ = expr e  in   Printf.fprintf file ";\n"
    | PrimitiveDeclaration(p, s, e) ->  (* Type, name, optional value *)
        let _ =
       (match p with
              Int ->  Hashtbl.add env.primitives s "int"
            | Char ->  Hashtbl.add env.primitives s "char"
            | Bool ->  Hashtbl.add env.primitives s "bool"
            | Float->  Hashtbl.add env.primitives s "float"
            | String ->  Hashtbl.add env.primitives s "string"
       ) in
      ( match p with
          Int ->
              ( match e with
                  | None -> Printf.fprintf file "int " ^ s ^ ";"
                  | (Some e1) -> Printf.fprintf file "int " ^ s ^ " = " ^ string_of_expr e1 ^ ";\n")
       | Char ->
```

```
            ( match e with
            | None -> Printf.fprintf file "char " ^ s ^ ";"
            | (Some e1) -> Printf.fprintf file "char " ^ s ^ " = " ^ string_of_expr e1 ^ ";\n")
        | Float ->
            ( match e with
            | None -> Printf.fprintf file "float " ^ s ^ ";"
            | (Some e1) -> Printf.fprintf file "int " ^ s ^ " = " ^ string_of_expr e1 ^ ";\n")
        | Bool ->
            ( match e with
            | None -> Printf.fprintf file "bool " ^ s ^ ";"
            | (Some e1) -> Printf.fprintf file "bool " ^ s ^ " = " ^ string_of_expr e1 ^ ";\n")
        | String ->
            ( match e with
            | None -> Printf.fprintf file "string " ^ s ^ ";"
            | (Some e1) -> Printf.fprintf file "string " ^ s ^ " = " ^ string_of_expr e1 ^ ";\n"))
     (* Type, name, size, optional values *)
     | ArrayDeclaration(s1, s2, e1, e2) ->
       let _ =  Hashtbl.add env.arrays s1 (s2, e1) in
            if (e2 == None)
         then Printf.fprintf file s1 ^ " " ^ s2 ^ "[" ^ string_of_expr e1 ^ "];\n"
            else if (e2 ==  (Some e))
         then  Printf.fprintf file s1 ^ " " ^ s2 ^ "[" ^ string_of_epr e1 ^ "] = "
                    ^ "{" ^ String.concat ", "  List.map (fun x -> string_of_expr x) e ^ "};\n"
         else ()
     | Attribute(s, p) -> (* Attribute name, type *)
      if (p == Int) then let _ =  Printf.fprintf file "typedef int " ^ s ^ ";" in Hashtbl.add
env.attributes s "int"
      else if (p == Char) then let _ =   Printf.fprintf file "typedef char " ^ s ^ ";"  in Hashtbl.add
env.attributes s "char"
      else if (p == Float) then let _ =   Printf.fprintf file "typedef float " ^ s ^ ";" in Hashtbl.add
env.attributes s "float"
      else if (p == Bool) then let _ =   Printf.fprintf file "typedef bool " ^ s ^ ";" in Hashtbl.add
env.attributes s "bool"
      else if (p == String) then let _ =   Printf.fprintf file "typedef char * " ^ s ^ ";" in Hashtbl.add
env.attributes s "string"
      else ()
     (* Name, list of attribute names - part prototype definition *)
     | Prototype(s, l)  -> let _ = List.iter (fun x -> if (Hashtbl.mem env.attributes x != true) then
                                        raise (Failure ("undefined attribute")) else () ) l
                       in let _ =  Printf.fprintf file "struct " ^ s ^ " { " ^ "attributes"  ^ "};"
                              in Hashtbl.add env.prototypes s l
      (* Head, list of RHS symbols - constraint definition *)
     | Constraint(s, l) ->
       if (Hashtbl.mem env.constraints s)
       then raise (Failure ("constrained is already defined"))
       else
         let _ = Printf.fprintf file "char *" ^ s ^ " [] [" ^ string_of_int List.length l ^ "] = { " ^
              String.concat ", " List.map (fun x -> "\"" ^ x ^ "\"") l ^ "};\n" ^
              (* append to linked list*)
```

```
                "index = index + 1;\n" ^
                "constraint_list = (struct constraint *)malloc(sizeof(struct constraint));\n" ^
                  "constraint_list->index = index;\n" ^
                  "memcpy(constraint_list->constraint_array, " ^ constraint_array ^ ", sizeof(" ^
constraint_array ^ "));\n" ^
                  "constraint_list->next = NULL;\n" ^
                  "constraint_list->prev = NULL;\n"
        in Hashtbl.add env.constraints s l

     | Break -> Printf.fprintf file "\tbreak;\n"
     | Continue -> Printf.fprintf file "\tcontinue;\n"
     | Return e     ->  Printf.fprintf file "\treturn" ^ string_of_expr e  ^ ";\n"
     | If (p, t, f) ->
        ( match f with
          Block [] -> let _ =  Printf.fprintf file "if ("; expr p; Printf.fprintf file ") \n" in stmt t
          | Block s  -> let _ =  Printf.fprintf file "if (" ^ string_of_expr p ^  ") \n"
                     in let _ =  stmt t
                            in let _ =  Printf.fprintf file "else\n"
                              in stmt s
       )
     | For (e1, e2, e3, b) ->
         stmt (Block([Expr(e1); While(e2, Block([b; Expr(e3)]))]))
     | While (e, b) ->
        let _ = Printf.fprintf file "while (" string_of_expr p ^  ") \n"
        in stmt b

   in   let env = { primitives = Hashtbl.create 100;
                  attributes = Hashtbl.create 100;
                  arrays = Hashtbl.create 100;
                  prototypes = Hashtbl.create 100;
                  parts = Hashtbl.create 100;
                  composites = Hashtbl.create 100;
                  functions = Hashtbl.create 100;
                  builtins = ["validate";"printSequence";"printDiagram";"generateMarkup"];
                  constraints = Hashtbl.create 100; }  in

          (* Includes, global definitions etc *)
          let entry_function () =
          Printf.fprintf file
            "#include <gtk/gtk.h>\n" ^
            "#include <stdio.h>\n" ^
            "#include <stdlib.h>\n" ^
            "#include \"builtins.h\"\n" ^
            "#include <string.h>\n" ^
          "extern int validate(struct constraint *, char **);\n" ^
          "extern void printDiagram(char **);\n" ^
            "extern void printSequence(char *);\n" ^
            "extern void printMarkup(const char *, char *);\n" ^
            "struct constraint * constraint_list;\n" ^
```

```
             "int index = 0;\n"
           in

           (* Compile the functions *)
           let _ = entry_function () in List.map (fun x -> process env x) allfunctions
```

biosyn.ml

```
type action = Ast | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);
                              ("-c", Compile) ]
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  match action with
    Ast -> let listing = Ast.string_of_program program
         in print_string listing
  | Compile ->  Compile.translate program
```

Example Programs

assembly.source – illustrates how to build and validate an assembly

```
// declare all attributes that can be properties of standard parts
Attribute Sequence string;
Attribute Name string ;
Attribute Compatibility string;
Attribute Strength float;

// part prototypes are declared prior to instantiation, each part uses one or more attributes
// technical definitions for the parts can be found in the glossary

// define the prototype for promoters
Part Promoter(Name, Sequence, Compatibility, Strength);
```

```
}

// define the prototype for an RBS
Part RBS(Name, Sequence, Compatibility);

// define the prototype for a terminator
Part Terminator(Name, Sequence, Compatibility);

// define the prototype for a cistron
Part Cistron(Name, Sequence, Strength);

// define the prototype for a cassette
Part Cassette(Name, Sequence, Compatibility);

// define the prototype for a gene Part Gene(Name, Sequence);
// define the prototype for a prefix Part Prefix(Name);
// define the prototype for a suffix Part Suffix(Name);
// instantiate the BioBricks promoter BBa_I14018
Promoter Bba_I14018(“BBa_I14018”, “tgtaagtttatacataggcgagtactctgttatgg”, “RFC21”, 0.5);

// instantiate the BioBricks RBS Bba_J63003
RBS Bba_J63003(“BBa_J63003”, “cccgccgccaccatggag”, “RFC21”);

// instantiate the BioBricks terminator BBa_B1002
Terminator Bba_B1002(“BBa_B1002”, “cgcaaaaaaccccgcttcggcggggttttttcgc”, “RFC21”);

// declare the constraints against which the assembly is parsed and validated
Start → <PlasmidBackbone><Prefix><Cassette><Suffix>;
Cassette → <Promoter><Cistron><Terminator>;
Cistron → <RBS><Gene>;
Cistron → <Cistron><Cistron>;
Terminator → <Terminator><Terminator>;
Gene → <Gene><Gene>;

// create an assembly, represented by a composite; composites may contain other composites
Composite Assembly1( Bba_I14018, Bba_J63003, Bba_B1002);

// validate the assembly against the declared constraints and then print sequence, print
diagram and
// generate markup

if (Assembly1.validate())
{
  Assembly1.printSequence();
  Assembly1.printDiagram();
  Assembly1.generateMarkup(“/usr/local/home/user1/assembly1markup.xml”);
}
```

primes.source – illustrates the use of the language for general algorithms

```
// Find primes upto 100
// Demonstrates the use of language constructs to implement algorithms

int n = 100;
primes(n);

int primes(int n)
{
        int  guess;
        int factor;


        print(”Find primes ” );
        print(1);
        print("\n");
        print(2);
        print("\n");
        print(2);
        print("\n");
        int remainder;
        guess = 5;
        while ( guess <= n )
         {
                 factor = 3;
                 remainder = guess - (guess / factor) * factor;
                 while ( (factor*factor < guess) && (remainder != 0 )
                 {
                          factor = factor + 2;
                }
                remainder = guess - (guess / factor) * factor;
                if (remainder != 0)
                {
                         print ( guess);
                         print("\n");
                }
                guess = guess +  2;
        }
}
```