JOT

A Musical Notation Language

Programmers Reference Manual

Brad Helstrom

August 13, 2010

for

COMS4115 - Programming Languages and Translators
Dr. Stephen Edwards
Columbia University

Table of Contents

1. Introduction

JOT is a compiler with which a composer can quickly represent a musical idea or collaborate for new ideas. The musical language will be intuitive to the composer and will have the capability to grow into scoring musical pieces for instruments, no matter what key the music will need to be expressed. The language will be built around notes, intervals and motifs which can be manipulated to create transposable musical notation.

This manual will walk you through the basic building blocks: types, operators and expressions, through the construction of motifs and songs, which can then be transposed. When the music is ready, it can be output to a standard text file which can be shared with others for collaboration.

2. Reference Manual

2.1. Expressions and Operators

2.1.1. Expressions

The expressions that are accepted by JOT include:

- *int*
- *rhythm*
- *note*
- *motif*

Variables in Jot will be expressed by using a series of lower case, upper case, numbers and underscores. The variables must begin with a lower case letter and can contain any combination of letters, numbers and underscores. The variables assigned to the value they express can not be any of the reserved words in the list below.

get	*for*	*while*
if	*else*	*int*
note	*rhythm*	*motif*
	transpose	

2.1.1.1. Accidentals, Notes and Octaves

The fundamental unit in this compiler is the note (with an optional accidental). It will be defined by letter, in keeping with musical notation.

accidental -> *# | b | ε*
octave -> *1 | 2 | 3 | 4 | 5 | 6 | 7 | ε*
note -> *octave (A | B | C | D | E | F | G) accidental*
rhythm -> *1 | 2 | 4 | 8 | 16 | 32 | ε*

Examples

```
note middle_c ;
note walking_bass_line;
```

Notes can be added, incremented by integer values representing chromatic steps.

2.1.1.2. Rhythms

The current version of Jot allows notes to be divided along duple meter, or by twos. The following notation will be used to associate a duration with a note to denote a note – rhythm combination. The notation is as follows:

SX = Sixteenth Note	
EG = Eighth Note	*DE = Dotted Eighth*
QT = Quarter Note	*DQ = Dotted Quarter*
HF = Half Note	*DH = Dotted Half*

The rhythms can be added, subtracted, divided and multiplied to create new durations and to subdivide notes for variations.

2.1.1.3. Motifs

Motifs allow the programmer to string together notes to define a snippet of music (i.e. a chorus, a verse, a motif, etc.).

The stringing together into a motif is achieved by using a 'add' operator (defined in the following section). Motifs can also be constructed of other motifs. Here are examples of constructing motifs.

Example:

```
mary_in = [4E:QT , 4D:QT , 4C:QT , 4D:QT , 4E:QT , 4E:QT , 4E:HF];
mary_out = [4D:QT , 4D:QT , 4E:QT , 4D:QT , 4C:HF];
mary_lamb = [mary_in add 4D:QT , 4D:QT , 4D:HF ,
                        4E:QT , 4G:QT , 4G:HF add mary_in add mary_out];
```

2.1.2. Operators

Operations that can be performed by JOT include:

+	-	/
*	>	<
=	>=	<=
	()	
add	*transpose*	*repeat*

In JOT, a line end will be marked with a semicolon ';'

Example:

```
breakin = [1A, QT : 1B, QT : 1C, QT : 1A, EG : 1B, QT : 1C, QT];
```

2.1.2.1. Motif add Operator

The 'add' will be used to sequence or add together two motifs or a note or chord with a motif. The result will be a longer motif, which eventually can be the length of the song being jotted down as shown in example in section 2.1.1.3 Motifs.

2.1.2.2. Motif insert Operator

It is possible to add notes to any space between notes in a motifs. Notes are inserted into a motif in the location specified by the space number. The number will denotes the space in which to add the note.

The proper usage of add will be as follows:

note insert (space number) motif;

- space number begins with the first space ahead of the first note and counts sequentially from there.

Example: Space Number

```
                    1       2       3       4       5       6
                    |       \/      \/      \/      \/      \/
blues_scale =       4A:EG,  4C:EG,  4D:EG,  4E:EG,  4G:EG,  5A:EG;

pent_blues_scale =  4Eb insert 4 blues_scale;  /* 4A 4C 4D 4Eb 4E 4G 5A */
```

2.1.2.3. transpose

The ability to transpose a piece quickly is one of the key advantages of jotting down music in this form. Jot will provide the ability to transpose motifs by an interval defined through the function transpose. The interval will be defined in musical terms with the following notation:

transpose motif int;

- int - Integer value will define the interval the transposition can take.

Examples:

```
riff = [3G:QT,  4C:QT, 4D:EG];
verse = [riff add riff add (transpose riff 7) add riff]; /* transpose up a 4th */
```

2.1.3. Functions

2.1.4. Statements and Conditionals

2.1.4.1. for

2.1.4.2. while

The ‘while’ operator allows the composer to loop through certain patterns and make decisions about note selections or to short cut scalar runs. To create the while loop, the statement is phrased as follows:

while(condition) {}

The brackets enclose the result condition behavior.

Examples:

```
/* Puts a top limit on a walking bass line */
while (running_base < 2A#)
    {
        walk_around ();
    }
```

2.1.4.3. if else

The ‘if’ operator allows the composer to set up constraints the program will use to construct new motifs. They can be used to establish interval restrictions, rhythmic repetitions, etc. They will follow the form:

if (condition) {} else {}

The brackets enclosure the result behavior.

Examples:

```
/* Gets a new note if the interval is too large */
if (new_note – old_note  > 5)
    {
        get new_note();
    }
```

3. Project Plan

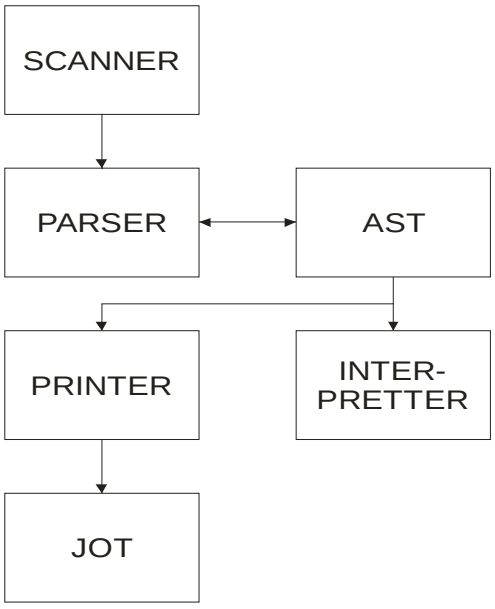
As I am working solo on this project, the assignments are made a bit easier to delegate. The project plan is as follows:

June – Familiarize with OCAML
July – Design the framework for JOT
- Work to develop a representation of the notes
- Get basic functionality (arithmetic) working between notes and integers

August – Implementation, Cleanup and Testing
- Develop further features and test for the compiler

4. Architectural Design

The architectural design of JOT is patterned after the 'microc' compiler/translator described in class. Many of the elements in the microc compiler have a place in JOT. The remaining pieces of JOT were added to the parser, scanner, ast, printer, interpreter and to the jot programs. The architecture is as follows:



5. Test Plan

The test plan is comprehensive and will attempt to catch most of the errors that can be issued by the compiler. The following tests were run on Jot to test its functionality. A lot of functionality did not make it into the compiler due to a lack of time. All tests passed successfully.

```
 Volume Serial Number is 320D-180E

 Directory of C:\Documents and Settings\Brad Helstrom\workspace\Jotter\src\Test
Files

08/12/2010  02:59 PM               183 jot-gcd.jot
08/12/2010  03:02 PM               225 jot-global1.jot
08/13/2010  04:21 PM                60 note-if1.jot
08/13/2010  04:30 PM                58 note-if2.jot
08/13/2010  03:18 PM               135 note-while1.jot
08/13/2010  03:29 PM                74 note-math3.jot
08/13/2010  03:36 PM               121 note-for1.jot
08/12/2010  08:40 PM                29 note-math1.jot
08/12/2010  08:55 PM                69 note-math2.jot
08/12/2010  08:59 PM                83 rhythm-math1.jot
08/13/2010  03:02 PM               125 note-function.jot
08/13/2010  04:10 PM                55 note-variable.jot
08/13/2010  04:24 PM                76 rhythm-if1.jot
08/13/2010  04:33 PM                70 rhythm-if2.jot
08/13/2010  04:43 PM               416 jot-ops1.jot
08/13/2010  05:02 PM               195 rhythm-function1.jot
              16 File(s)          1,974 bytes
               0 Dir(s)   3,459,006,464 bytes free
```

6. Lessons Learned

I've been in college coursework for well over fifteen years and this without question is the most difficult project I have worked on. It contained a confluence of new experiences: a new language, new tools, new concepts in software development and new processes. It took me a long time to grasp that while I was creating a program, it was not like any other program I have written in the past. It was a program with which further programs would be written. It was not necessarily written to perform a function, rather to provide an apparatus with which other computer scientists and composers could write a program to perform a function or to solve a problem. It was a level of abstraction that I simply was not ready for.

I have struggled mightily in using the tools and getting the basics built. My original plan was to have a working compiler running by the end of July. I did not get a successful first build working until the end of the first week of August. This did not give me a lot of time to fill in as much as I would have liked. My key take away from this is to get the tools in place and understood early in the process. Also, compile some smaller compilers to get a feel for the compilation and testing. I was using the OCAML toplevel to get a familiarity with OCAML. I now know that I should have gotten the tools in place, namely an IDE, and used the tools to help develop my understanding of OCAML. I could have tackled two major learning curves at the same time.

7. Appendix

7.1. Scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { OPNPR }
| ')'      { CLSEPR }
| '{'      { OPNBR }
| '}'      { CLSEBR }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| '#'      { SHARP }
| 'b'      { FLAT }
| "SX"     { SIXTEENTH }
| "EG"     { EIGHTH }
| "QT"     { QUARTER }
| "HF"     { HALF }
| "DE"     { DEIGHTH }
| "DQ"     { DQUARTER }
| "DH"     { DHALF }
| "get"    { GET }
| "if"     { IF }
| "note"   { NOTE }
| "rhythm" { RHYTHM }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "motif"  { MOTIF }
| "int"    { INT }
| "transpose"           { TRANSPOSE }
| "jot"                 { JOT }
| ['0'-'9']+ as lxm   { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| ['0'-'9']['A'-'G'] as entr           (* suedo-base12 *)
                        { BASE(((int_of_char entr.[0] - 48) * 12 ) +
                           ((int_of_char entr.[1] - 64) * 16 / 10)) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

7.2. Parser.mly

```
%{ open Ast %}

%token SEMI OPNPR CLSEPR OPNBR CLSEBR COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE FOR WHILE INT NOTE RHYTHM MOTIF
%token PLUS MINUS STAR SLASH ASSIGN SHARP FLAT
%token HALF QUARTER EIGHTH SIXTEENTH
%token DHALF DQUARTER DEIGHTH
%token GET TRANSPOSE JOT
%token <int> LITERAL BASE
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc SHARPD
%nonassoc FLATD

%left ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

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

fdecl:
   ID OPNPR formals_opt CLSEPR OPNBR vdecl_list stmt_list CLSEBR
                           { { fname = $1;
                               formals = $3;
                               locals = List.rev $6;
                               body = List.rev $7 } }

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

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

vdecl_list:
    /* nothing */          { [] }
  | vdecl_list vdecl       { $2 :: $1 }

vdecl:
    INT ID SEMI            { $2 }
  | MOTIF ID SEMI          { $2 }
  | NOTE ID SEMI           { $2 }
  | RHYTHM ID SEMI         { $2 }

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

stmt:
    expr SEMI              { Expr($1) }
  | RETURN expr SEMI       { Return($2) }
  | OPNBR stmt_list CLSEBR { Block(List.rev $2) }
```

```
  | IF OPNPR expr CLSEPR stmt %prec NOELSE
                            { If($3, $5, Block([])) }
  | IF OPNPR expr CLSEPR stmt ELSE stmt
                            { If($3, $5, $7) }
  | FOR OPNPR expr_opt SEMI expr_opt SEMI expr_opt CLSEPR stmt
                            { For($3, $5, $7, $9) }
  | WHILE OPNPR expr CLSEPR stmt
                            { While($3, $5) }

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

note:
    BASE                    { $1 }
  | BASE FLAT  %prec FLATD  { $1 - 1 }
  | BASE SHARP %prec SHARPD { $1 + 1 }

rhythm:
    SIXTEENTH               { 1 }
  | EIGHTH                  { 2 }
  | DEIGHTH                 { 3 }
  | QUARTER                 { 4 }
  | DQUARTER                { 6 }
  | HALF                    { 8 }
  | DHALF                   { 12 }

expr:
    LITERAL                 { Literal($1) }
  | ID                      { Id($1) }
  | rhythm                  { Note($1) }
  | note                    { Rhythm($1) }
  | expr PLUS   expr        { Binop($1, Add,      $3) }
  | expr MINUS  expr        { Binop($1, Subtract, $3) }
  | expr TIMES  expr        { Binop($1, Multipy,  $3) }
  | expr DIVIDE expr        { Binop($1, Divide,   $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) }
  | ID ASSIGN expr          { Assign($1, $3) }
  | ID OPNPR actuals_opt CLSEPR { Call($1, $3) }
  | OPNPR expr CLSEPR       { $2 }

note_set:
   note COMMA rhythm       { $1, $3 }

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

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

7.3. Ast.mli

```
%{ open Ast %}

%token SEMI OPNPR CLSEPR OPNBR CLSEBR COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE FOR WHILE INT NOTE RHYTHM MOTIF
%token PLUS MINUS STAR SLASH ASSIGN SHARP FLAT
%token HALF QUARTER EIGHTH SIXTEENTH
%token DHALF DQUARTER DEIGHTH
%token GET TRANSPOSE JOT
%token <int> LITERAL BASE
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc SHARPD
%nonassoc FLATD

%left ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

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

fdecl:
   ID OPNPR formals_opt CLSEPR OPNBR vdecl_list stmt_list CLSEBR
                           { { fname = $1;
                               formals = $3;
                               locals = List.rev $6;
                               body = List.rev $7 } }

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

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

vdecl_list:
    /* nothing */          { [] }
  | vdecl_list vdecl       { $2 :: $1 }

vdecl:
    INT ID SEMI            { $2 }
  | MOTIF ID SEMI          { $2 }
  | NOTE ID SEMI           { $2 }
  | RHYTHM ID SEMI         { $2 }

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

stmt:
    expr SEMI              { Expr($1) }
  | RETURN expr SEMI       { Return($2) }
  | OPNBR stmt_list CLSEBR { Block(List.rev $2) }
```

```
  | IF OPNPR expr CLSEPR stmt %prec NOELSE
                            { If($3, $5, Block([])) }
  | IF OPNPR expr CLSEPR stmt ELSE stmt
                            { If($3, $5, $7) }
  | FOR OPNPR expr_opt SEMI expr_opt SEMI expr_opt CLSEPR stmt
                            { For($3, $5, $7, $9) }
  | WHILE OPNPR expr CLSEPR stmt
                            { While($3, $5) }

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

note:
    BASE                    { $1 }
  | BASE FLAT  %prec FLATD  { $1 - 1 }
  | BASE SHARP %prec SHARPD { $1 + 1 }

rhythm:
    SIXTEENTH               { 1 }
  | EIGHTH                  { 2 }
  | DEIGHTH                 { 3 }
  | QUARTER                 { 4 }
  | DQUARTER                { 6 }
  | HALF                    { 8 }
  | DHALF                   { 12 }

expr:
    LITERAL                 { Literal($1) }
  | ID                      { Id($1) }
  | rhythm                  { Note($1) }
  | note                    { Rhythm($1) }
  | expr PLUS   expr        { Binop($1, Add,      $3) }
  | expr MINUS  expr        { Binop($1, Subtract, $3) }
  | expr TIMES  expr        { Binop($1, Multipy,  $3) }
  | expr DIVIDE expr        { Binop($1, Divide,   $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) }
  | ID ASSIGN expr          { Assign($1, $3) }
  | ID OPNPR actuals_opt CLSEPR { Call($1, $3) }
  | OPNPR expr CLSEPR       { $2 }

note_set:
   note COMMA rhythm       { $1, $3 }

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

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

7.4. Printer.ml

```
open Ast

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Id(s) -> s
  | Note(n) -> string_of_int (n/12) ^ "octave"
  | Rhythm(r) -> (match r with
        1 -> "SX"
      | 2 -> "EG"
      | 3 -> "DE"
      | 4 -> "QT"
      | 6 -> "DQ"
      | 8 -> "HF"
      | 12 -> "DH"
      | _ -> "Error, not a valid rhythm")
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
        Add -> "+" | Subtract -> "-" | Multipy -> "*" | Divide -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=") ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

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

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

let string_of_fdecl fdecl =
  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 (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

7.5. Interpretter.ml

```
open Ast

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

exception ReturnException of int * int NameMap.t

(* Main entry point: run a program *)

let run (vars, funcs) =
  (* Put function declarations in a symbol table *)
  let func_decls = List.fold_left
      (fun funcs fdecl -> NameMap.add fdecl.fname fdecl funcs)
      NameMap.empty funcs
  in

  (* Invoke a function and return an updated global symbol table *)
  let rec call fdecl actuals globals =

    (* Evaluate an expression and return (value, updated environment) *)
    let rec eval env = function
        Literal(i) -> i, env
      | Note(n) -> n, env
      | Rhythm(r) -> r, env
      | Noexpr -> 1, env (* must be non-zero for the for loop predicate *)
      | Id(var) ->
          let locals, globals = env in
          if NameMap.mem var locals then
            (NameMap.find var locals), env
          else if NameMap.mem var globals then
            (NameMap.find var globals), env
          else raise (Failure ("undeclared identifier " ^ var))
      | Binop(e1, op, e2) ->
          let v1, env = eval env e1 in
           let v2, env = eval env e2 in
          let boolean i = if i then 1 else 0 in
          (match op with
            Add -> v1 + v2
          | Subtract -> v1 - v2
          | Multipy -> v1 * v2
          | Divide -> v1 / v2
          | Equal -> boolean (v1 = v2)
          | Neq -> boolean (v1 != v2)
          | Less -> boolean (v1 < v2)
          | Leq -> boolean (v1 <= v2)
          | Greater -> boolean (v1 > v2)
          | Geq -> boolean (v1 >= v2)), env
      | Assign(var, e) ->
          let v, (locals, globals) = eval env e in
          if NameMap.mem var locals then
            v, (NameMap.add var v locals, globals)
          else if NameMap.mem var globals then
            v, (locals, NameMap.add var v globals)
          else raise (Failure ("undeclared identifier " ^ var))
      | Call("print", [e]) ->
          let v, env = eval env e in
          print_endline (string_of_int v);
          0, env
      | Call(f, actuals) ->
          let fdecl =
            try NameMap.find f func_decls
            with Not_found -> raise (Failure ("undefined function " ^ f))
          in
          let actuals, env = List.fold_left
              (fun (actuals, env) actual ->
                let v, env = eval env actual in v :: actuals, env)
```

```
            ([], env) actuals
        in
        let (locals, globals) = env in
        try
          let globals = call fdecl actuals globals in 0, (locals, globals)
        with ReturnException(v, globals) -> v, (locals, globals)
  in

  (* Execute a statement and return an updated environment *)
  let rec exec env = function
      Block(stmts) -> List.fold_left exec env stmts
    | Expr(e) -> let _, env = eval env e in env
    | If(e, s1, s2) ->
        let v, env = eval env e in
        exec env (if v != 0 then s1 else s2)
    | While(e, s) ->
        let rec loop env =
          let v, env = eval env e in
          if v != 0 then loop (exec env s) else env
        in loop env
    | For(e1, e2, e3, s) ->
        let _, env = eval env e1 in
        let rec loop env =
          let v, env = eval env e2 in
          if v != 0 then
            let _, env = eval (exec env s) e3 in
            loop env
          else
            env
        in loop env
    | Return(e) ->
        let v, (locals, globals) = eval env e in
        raise (ReturnException(v, globals))
  in

  (* Enter the function: bind actual values to formal arguments *)
  let locals =
    try List.fold_left2
        (fun locals formal actual -> NameMap.add formal actual locals)
        NameMap.empty fdecl.formals actuals
    with Invalid_argument(_) ->
      raise (Failure ("wrong number of arguments passed to " ^ fdecl.fname))
  in
  (* Initialize local variables to 0 *)
  let locals = List.fold_left
      (fun locals local -> NameMap.add local 0 locals) locals fdecl.locals
  in
  (* Execute each statement in sequence, return updated global symbol table *)
  snd (List.fold_left exec (locals, globals) fdecl.body)

(* Run a program: initialize global variables to 0, find and run "main" *)
in let globals = List.fold_left
    (fun globals vdecl -> NameMap.add vdecl 0 globals) NameMap.empty vars
in try
  call (NameMap.find "main" func_decls) [] globals
with Not_found -> raise (Failure ("did not find the main() function"))
```

7.6. Jot.ml

```
let print = false

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  if print then
    let listing = Printer.string_of_program program in
    print_string listing
  else
    ignore (Interpret.run program)
```

7.7. Makefile.bat

```
cd \
cd Documents and Settings\Brad Helstrom\workspace\jotter\src

ocamllex   scanner.mll
ocamlyacc  parser.mly
ocamlc -c  ast.mli
ocamlc -c  parser.mli
ocamlc -c  parser.ml
ocamlc -c  scanner.ml
ocamlc -c  printer.ml
ocamlc -c  interpret.ml
ocamlc -c  jotter.ml
ocamlc -o  jotter.exe parser.cmo scanner.cmo printer.cmo interpret.cmo jotter.cmo
pause
cls

ocamldep *.ml *.mli

pause

cd Test Files
jot_test
```

7.8. Jot_test.bat

```
cls

@echo off

echo ..
echo Test Script for testing the Jot compiler
echo Tests with include simple math, conditional statements
echo Notes, Note pairs, motifs and transpose functions
echo ..

for %%X in (*.jot) do (
        jotter < %%~nX.jot > %%~nX.ans
        fc %%~nX.out %%~nX.ans > %%nX.diff
        IF ERRORLEVEL 1 echo %%~nX ---------- TEST FAILED
        IF ERRORLEVEL 0 echo %%~nX TEST PASSED
        )
pause
```