Programming Languages & Translators: BachEnd (Final Project)

Audrey Acken, Eden Chung, Yahya ElGawady, Apurva Reddy, Emily Yin

BachEnd Motivation

- Musicians can write sheet music **efficiently**
- Current tools are either...
  - **Tedious** and **not typed**
  - **Complicated**
- Export code as a standard **sheet music PDF**
- BachEnd syntax is both **intuitive** and **flexible**

Language in one slide (song.bach)

```
NUMBER main () {
    NOTE xx!
    WRITE(NAME="My Song", TEMPO=100, CLEF=TREBLE, TIMESIGNATURE=(4,4), KEYSIGNATURE="C") {
      [C# E 2C_E_G]!

      xx = [4C 4C 4C 4C]!

      REPEAT(2) {
        xx!
      }

      TRANSPOSE(4) {
        xx!
      }
    }
}
```

Features

- Syntax
  - Functions, built-in features in ALL CAPS
    - Easily identifiable from user variables
    - Matches syntax of musical notes
  - “!” (exclamation mark) as line delimiter
    - Intuitive line endings for non-programmer musicians
- Built in musical functions
  - REPEAT(times)
  - TRANSPOSE (half_steps)
  - WRITE(notes)
- Everything must be declared within the MAIN function, and anything to be generated in the final PDF must be written within the WRITE function

Notes/Rests

The NOTE type represents a single musical note, it has the following properties:

- Properties:
  - PITCH:
    - A STRING representing the note name (ex "C#").
    - Values: Ab, A, A#, Bb, B, B#, Cb, C, C#, Db, D, D#, Eb, E, E#, Fb, F, F#, Gb, G, G#, R
  - OCTAVE:
    - The OCTAVE specifies the register of a note. BachEnd octaves are represented as a NUMBER, where middle C is typically 4.
    - We would specify the note B in the 5th octave as this: B5
  - LENGTH:
    - The LENGTH represents the length of a note.
    - BachEnd lengths are represented as a NUMBER, defined in powers of two, where 1 = a whole note, 2 = a half note, 4 = a quarter note, 8 = an eighth note, 16 = a sixteenth note, etc.).
    - We would specify the note B as a quarter note as this: 4B
- Putting it together, if we wanted to define the note B in the 5th octave as a quarter note, we would write 4B5

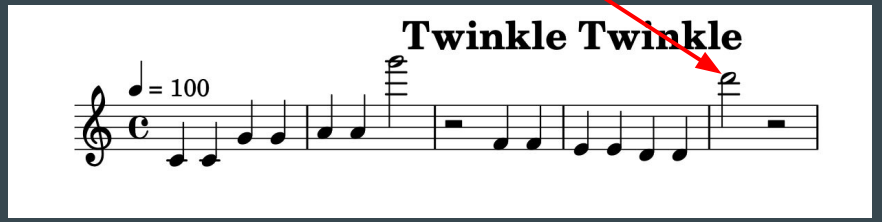
The REST type represents a musical rest. It is defined as R. Like NOTE, the REST type has a length property. If we wanted a quarter rest we would write 4R.

```
NUMBER main () {
    WRITE(NAME="Twinkle Twinkle", TEMPO=100) {
        [4C 4C 4G 4G 4A 4A 2G6 2R]
        [F F 4E 4E 4D 4D 2D6 2R]!
    }
}
```

If duration and octave are unspecified, they default to a quarter note (duration 4) at octave 4

Rest (half note timing)

One D note (half note timing), of octave 6

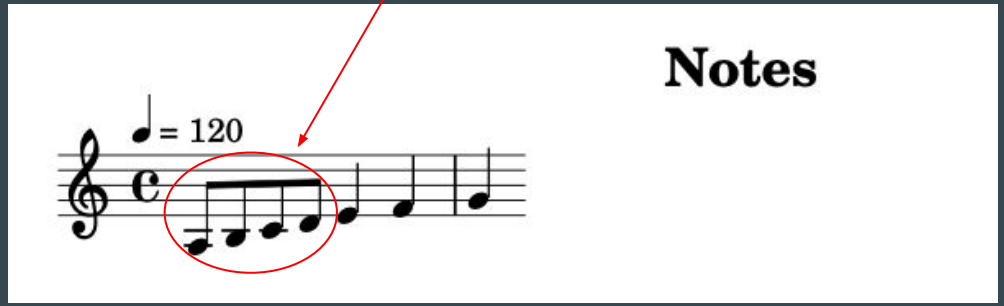


Combined Notes

- Grouping of notes with the same duration and/or octave

```
NUMBER main () {
    WRITE(NAME="Notes", TEMPO=120) {
    [8(A3 B3 C D) 4(E F G)]!
  }
}
```

Group of notes all with eighth note duration marked by ()



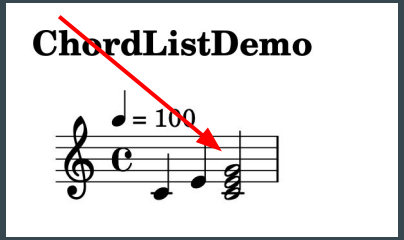
Chords

A CHORD is a combination of multiple NOTE values played simultaneously.

- Say you wanted to play a C major triad CHORD, It would be written as C_E_G.
- Like NOTE, you can specify PITCH, OCTAVE and LENGTH. Say we played a C major triad CHORD in the 3rd octave for an eighth note, we would write 8C_E_G3.
- The default value for OCTAVE is 4 and for LENGTH is 4.
- A CHORD may contain notes from more than one OCTAVE. Consider the C Major CHORD which would be written as C4_E4_G4_C5
- A CHORD may contain notes from more than one LENGTH. Consider this chord from the first measure of Beethoven's famous *Sonata No. 8 "Pathetique"*. This would be written as 4B_16G 16(E G E)

```
NUMBER main () {
  WRITE(NAME="ChordListDemo", TEMPO=100, CLEF=TREBLE, TIMESIGNATURE=(4,4), KEYSIGNATURE="C") {
    [C E 2C_E_G]!
  }
}
```

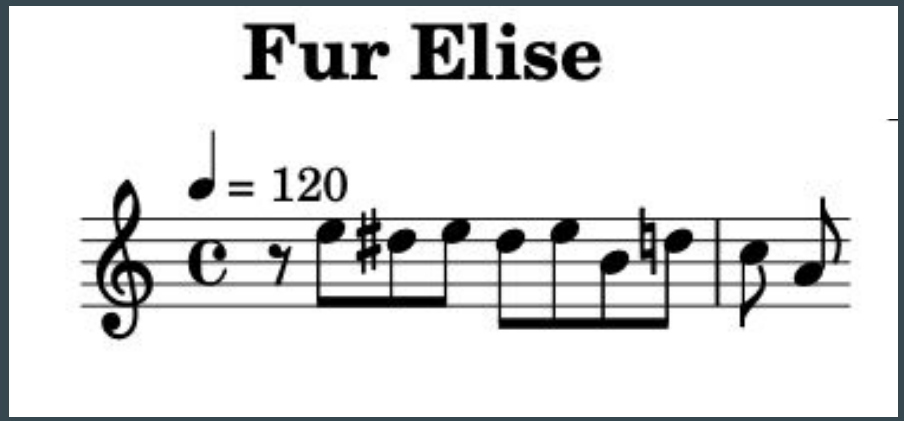
C major chord



Making a song

- WRITE needs the parameters NAME and TEMPO

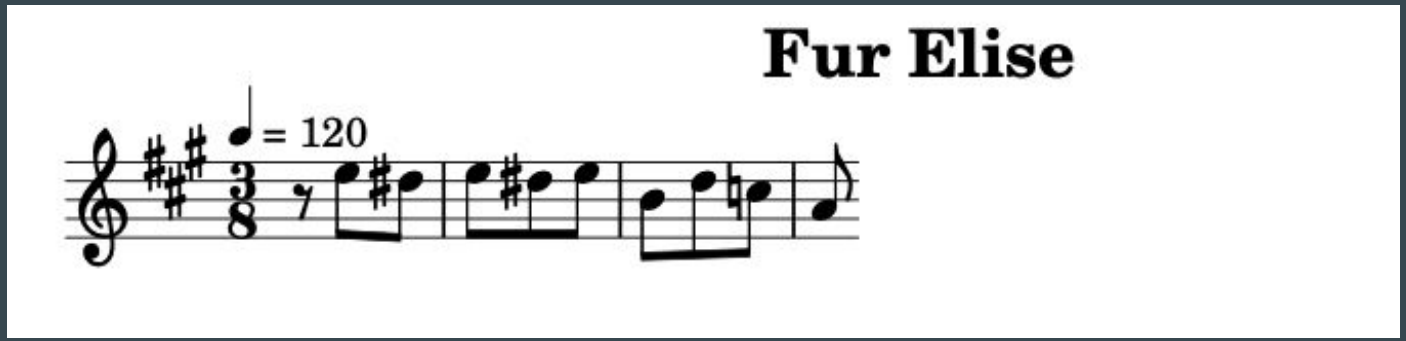
```
NUMBER main () {
    WRITE(NAME="Fur Elise", TEMPO=120) {
        [8R 8E5 8D#5 8E5 8D#5 8E5 8B4 8D5 8C5 8A4]!
    }
}
```



Making a song

- We can also add certain optional arguments, such as clef, time signature, and key signature
  - If omitted, it will default to treble clef with time signature (4,4), and key signature C major.

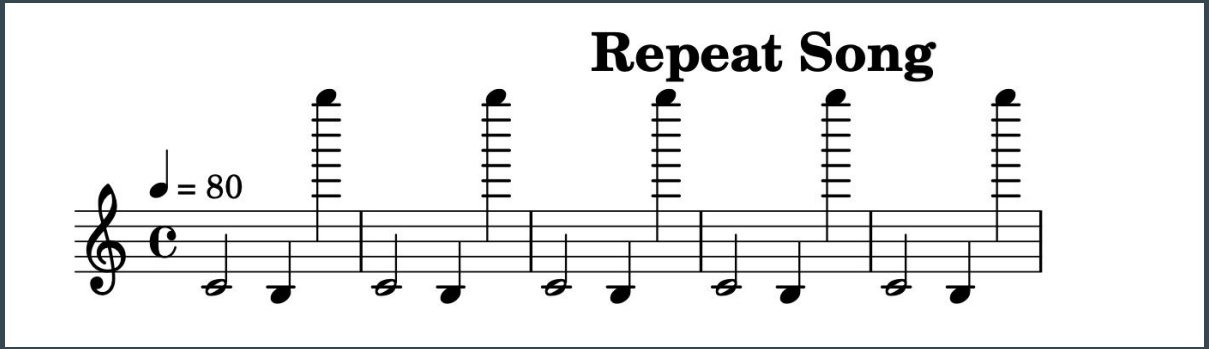
```
NUMBER main () {
    WRITE(NAME="Fur Elise", TEMPO=120, CLEF=TREBLE, TIMESIGNATURE=(3,8), KEYSIGNATURE="A") {
    [8R 8E5 8D#5 8E5 8D#5 8E5 8B4 8D5 8C5 8A4]!
  }
}
```



REPEAT(times)

- Repeat a sequence of notes a number of times

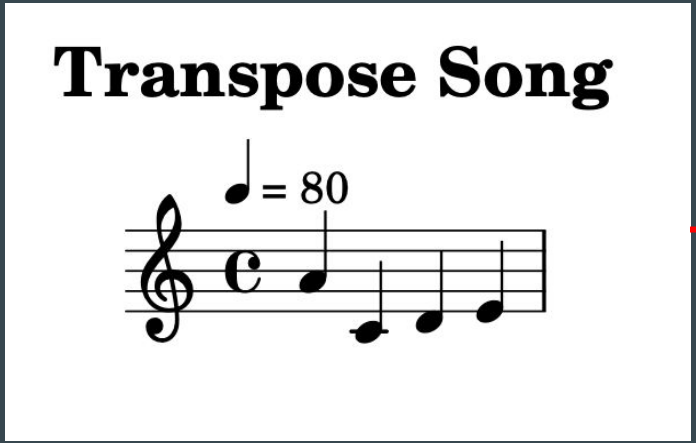
```
1  NUMBER main () {
2      WRITE(NAME="Repeat Song", TEMPO=80) {
3          REPEAT(5) {
4              [2C 4B3 G7]!
5          }
6      }
7  }
```



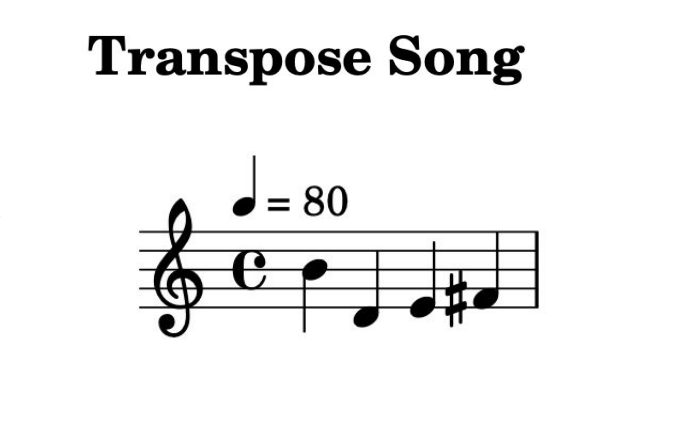
TRANSPOSE(half_steps)

- Shift sequence of notes up/down by a number of half steps

```
NUMBER main () {
    WRITE(NAME="Transpose Song", TEMPO=80) {
        TRANSPOSE(2) {
            [A]!
            [C D E]!
        }
    }
}
```

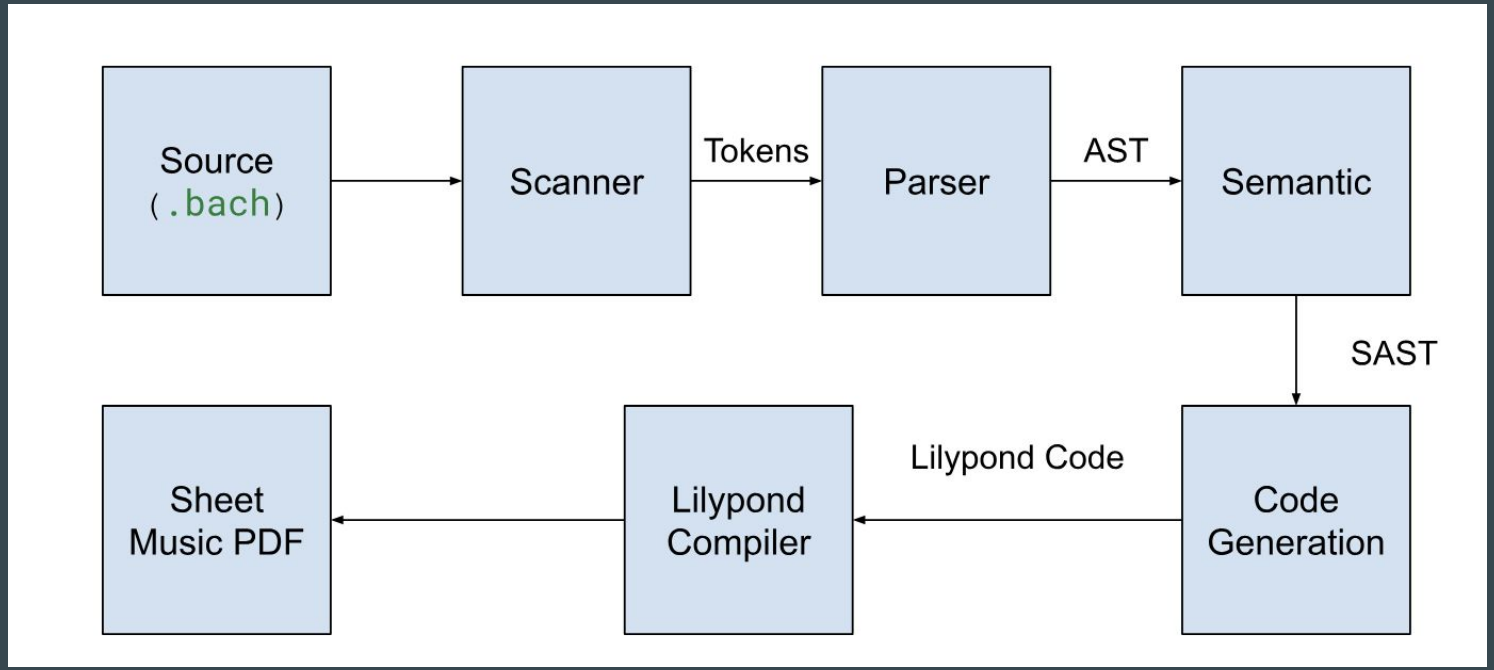


Each note shifted up 2 half steps



Demo

Architectural design



Scanner

```
let alpha = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let alphanumeric = ['a'-'z' 'A'-'Z' '0'-'9']
let whitespace = [' ' '\t' '\r' '\n']

let alpha_lower = ['a'-'z']
let id = alpha_lower alphanumeric+ (* must be at least 2 characters long*)
let INT = '-'? ('0' | (['1'-'9'] digit*))
let string_body = [^ '"' ]* (* allow strings to have spaces in them *)
let STRING = '"' string_body '"'

let number      = ['0'-'9']+
let octave_digit = ['1'-'8']
let accidental  = ['#' 'b']
let base_note   = ['A'-'G']
let note_char   = base_note accidental? | 'R'
let NOTE        = number? note_char octave_digit?
```

Scanner

```
| NOTE as lxm {
    (* break out prefix digits, pitch+accidental, and suffix digit *)
    let len, rest =
      let buf = Buffer.create 3 in
      let i = ref 0 in
      while !i < String.length lxm && lxm.[!i] >= '0' && lxm.[!i] <= '9' do
        Buffer.add_char buf lxm.[!i];
        incr i
      done;
      Buffer.contents buf, !i
    in
    let pitch_acc_oct = String.sub lxm rest (String.length lxm - rest)
    in
    (* if last char is digit, that's the octave *)
    let pitch, oct_str =
      let len_pao = String.length pitch_acc_oct in
      if len_pao > 1 && pitch_acc_oct.[len_pao-1] >= '1'
         && pitch_acc_oct.[len_pao-1] <= '8'
      then
        ( String.sub pitch_acc_oct 0 (len_pao-1)
        , String.make 1 pitch_acc_oct.[len_pao-1] )
      else
        ( pitch_acc_oct, "" )
    in
    let length = if len = "" then 4 else int_of_string len in
    let octave = if oct_str = "" then 4 else int_of_string oct_str in
    let n = { Ast.pitch; octave; length } in
    NOTELIT n
}
```

Scanner

```
| number? base_note ("_" base_note)+ octave_digit? as lxm {
  (* split off leading length digits *)
  let len_str, rest_idx =
    let buf = Buffer.create 4 and i = ref 0 in
    while !i < String.length lxm && lxm.[!i] >= '0' && lxm.[!i] <= '9' do
      Buffer.add_char buf lxm.[!i]; incr i
    done;
    Buffer.contents buf, !i
  in
  (* the pitches+octave suffix *)
  let body = String.sub lxm rest_idx (String.length lxm - rest_idx) in
  (* if last char is octave digit *)
  let pitch_part, oct_str =
    if body <> "" && (body.[String.length body - 1] >= '1' && body.[String.length body - 1] <= '8') then
      (String.sub body 0 (String.length body - 1), String.make 1 body.[String.length body - 1])
    else body, ""
  in
  let length = if len_str = "" then 4 else int_of_string len_str in
  let octave = if oct_str = "" then 4 else int_of_string oct_str in
  let pitches = Str.split (Str.regexp "_") pitch_part in
  let notes = List.map (fun p -> { Ast.pitch = p; octave; length }) pitches in
  CHORDLIT notes
}
```

Parser and AST

```
stmt:
  | typ ID ASSIGN expr EXCLAMATION { VDecl($1, $2, $4) }
  | LBRACE stmt_list RBRACE                { Block $2 }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | WHILE LPAREN expr RPAREN stmt           { While ($3, $5)  }
  /* return */
  | RETURN expr EXCLAMATION                        { Return $2      }
  | REPEAT LPAREN expr RPAREN stmt       { Repeat($3, $5) }
  | TRANSPOSE LPAREN expr RPAREN stmt       { Transpose($3, $5) }
  | WRITE LPAREN NAME ASSIGN STRING COMMA TEMPO ASSIGN LITERAL write_optional_args RPAREN stmt
    {
      let raw = $5 in
      let name = String.sub raw 1 (String.length raw - 2) in
      let (clef_opt, ts_opt, ks_opt) = $10 in
      WriteAttrs {
        name = name;
        tempo = $9;
        clef = clef_opt;
        timesig = ts_opt;
        keysig = ks_opt;
        body = $12;
      }
    }
  | expr EXCLAMATION                          { Expr $1      }
```

```
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"
  | Break -> "BREAK!\n"
  | Continue -> "CONTINUE!\n"
  | If(e, s1, s2) ->  "IF (" ^ string_of_expr e ^ ")\n" ^
                      string_of_stmt s1 ^ "ELSE\n" ^ string_of_stmt s2
  | While(e, s) -> "WHILE (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | For(x, y, z) -> "FOR (" ^ x ^ "IN " ^ string_of_expr y ^ ") " ^ string_of_stmt z
  | Repeat(x, s) -> "REPEAT (" ^string_of_expr x ^ ") " ^ string_of_stmt s
  | Print(x) -> "PRINT (" ^ string_of_expr x ^ ")!"
  | WriteAttrs { name; tempo; clef; timesig; keysig; body } ->
    let meta =
      Printf.sprintf "NAME=\"%s\", TEMPO=%d" name tempo ^
      (match clef with Some c -> ", CLEF=" ^ c | None -> "") ^
      (match timesig with Some (a, b) -> Printf.sprintf ", TIMESIGNATURE=(%d,%d)" a b | None -> "") ^
      (match keysig with Some k -> ", KEYSIGNATURE=\"" ^ k ^ "\"" | None -> "")
    in
    "WRITE(" ^ meta ^ ") " ^ string_of_stmt body
```

Parser and AST

```
note_list:
    NOTELIT                { [[ $1 ]]                }
  | CHORDLIT               { [ $1 ]                  }
  | NOTELIT note_list       { [ $1 ] :: $2            }
  | CHORDLIT note_list      { $1       :: $2          }

expr:
    LITERAL          { Literal($1)            }
  | BLIT             { BoolLit($1)            }
  | NOTELIT          { NoteLit($1)            }
  | CHORDLIT         { ChordLit($1)           }
  | ID               { Id($1)                 }
  | expr PLUS   expr { Binop($1, ADD,   $3)   }
  | expr MINUS  expr { Binop($1, SUB,   $3)   }
  | expr TIMES  expr { Binop($1, TIMES,   $3)   }
  | expr DIVIDE  expr { Binop($1, DIVIDE,   $3)   }
  | expr EQUAL     expr { Binop($1, EQUAL, $3)   }
  | expr NEQ    expr { Binop($1, NEQ, $3)     }
  | expr LEQ    expr { Binop($1, LEQ, $3)     }
  | expr GEQ    expr { Binop($1, GEQ, $3)     }
  | expr LT     expr { Binop($1, LT,  $3)   }
  | expr GT     expr { Binop($1, GT,  $3)   }
  | expr AND    expr { Binop($1, AND,   $3)   }
  | expr OR     expr { Binop($1, OR,    $3)   }
  | NOT expr         { Unop(NOT, $2)          }
  | ID ASSIGN expr   { Assign($1, $3)         }
  | LPAREN expr RPAREN { $2                   }
  /* call */
  | ID LPAREN args_opt RPAREN { Call ($1, $3)  }
  | LBRACKET note_list RBRACKET { NoteList($2)  }
```

```
let rec string_of_expr = function
    Literal l      -> string_of_int l
  | BoolLit true  -> "TRUE"
  | BoolLit false -> "FALSE"
  | StringLit s -> s
  | NoteLit n -> string_of_int n.length ^ n.pitch ^string_of_int n.octave
  | Id s          -> s
  | Binop (e1, o, e2) ->
    string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop (u, e) -> string_of_unop u ^ " " ^ string_of_expr e
  | Assign (v, e) -> v ^ " = " ^ string_of_expr e
  | TraitAssign (n, t, e) -> n ^ "." ^ t ^ " = " ^ string_of_expr e
  | Call (f, el)  ->
     f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | NoteList groups ->
   "["
   ^ String.concat " "
      (List.map
         (fun notes ->
           if List.length notes = 1 then
             (* single note group *)
             let n = List.hd notes in
             string_of_int n.length ^ n.pitch ^ string_of_int n.octave
           else
             (* chord group *)
             "<"
             ^ String.concat " "
                 (List.map (fun n ->
                      string_of_int n.length ^ n.pitch ^ string_of_int n.octave
                   ) notes)
             ^ ">"
         )
         groups
      )
   ^ "]"
```

Parser

```
write_optional_args:
    /* no extra args */ { (None, None, None) }

  | COMMA CLEF ASSIGN clef_val write_optional_args {
      let (clef, ts, ks) = $5 in
      (Some $4, ts, ks)
    }

  | COMMA TIMESIGNATURE ASSIGN LPAREN LITERAL COMMA LITERAL RPAREN write_optional_args {
      let (clef, ts, ks) = $9 in
      (clef, Some ($5, $7), ks)
    }

  | COMMA KEYSIGNATURE ASSIGN STRING write_optional_args {
      let raw = $4 in
      let key = String.sub raw 1 (String.length raw - 2) in
      let (clef, ts, _) = $5 in
      (clef, ts, Some key)
    }
```

SAST

```
(* semantically-checked expression *)
and sx =
  | SLiteral of int (* int literal *)
  | SBoolLit of bool (* boolean literal *)
  | SNoteLit of note  (* note literal *)
  | SNoteList of note list list
  | SChordLit  of note list
  | SId of string (* variable identifier *)
  | SAssign of string * sexpr (* variable assignment: string is var name, sexpr is expression *)
  | SBinop of sexpr * op * sexpr (* binary operator: left operand, operator, right operand *)
  | SCall of string * sexpr list (* function call: function name, list of arguments *)

(* A typed statement *)
type sstmt =
  | SBlock of sstmt list
  | SExpr of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SWhile of sexpr * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SPrint of sexpr
  | SRepeat of sexpr * sstmt
  | SWrite of sstmt
  | STranspose of sexpr * sstmt
  | SWriteAttrs of {
    name : string;
    tempo : int;
    clef : string option;
    timesig : (int * int) option;
    keysig : string option;
    body : sstmt;
  }
  (* return statement *)
  | SReturn of sexpr
```

Semantic - semantic.ml

AST to SAST: type checking, verifying declarations, correct variable usage

```
let rec check_stmt_list =function
    [] -> []
  | Block sl :: sl'  -> check_stmt_list (sl @ sl') (* Flatten blocks *)
  | s :: sl -> check_stmt s :: check_stmt_list sl
(* Return a semantically-checked statement i.e. containing sexprs *)
and check_stmt =function
  (* A block is correct if each statement is correct and nothing
     follows any Return statement.  Nested blocks are flattened. *)
    Block sl -> SBlock (check_stmt_list sl)
  | Expr e -> SExpr (check_expr e)
  | If(e, st1, st2) ->
    SIf(check_bool_expr e, check_stmt st1, check_stmt st2)
  | While(e, st) ->
    SWhile(check_bool_expr e, check_stmt st)
  | Repeat(e, st) ->
    let (t, e') = check_expr e in
    if t = Int then SRepeat ((t, e'), check_stmt st)
    else raise (Failure ("repeat requires an integer expression in " ^ string_of_expr e))
  | Write(stmt_block) -> SWrite (check_stmt stmt_block)
  | Print e ->
    let (t, e') = check_expr e in
    begin match t with
      | Int | Bool | Note -> SPrint(t, e') (* can modify this later*)
      | _ -> raise (Failure ("cannot print expression of type " ^ string_of_typ t ^ " in " ^ string_of_expr e))
    end
  | Return e ->
    let (t, e') = check_expr e in
    if t = func.rtyp then SReturn (t, e')
    else raise (
        Failure ("return gives " ^ string_of_typ t ^ " expected " ^
                 string_of_typ func.rtyp ^ " in " ^ string_of_expr e))
```

Code - irgen.ml

- Takes as input SAST from semant.ml
- Converts this input into Lilypond code

```
\header { title = "Transpose Song" }
\version "2.24.2"
\score { \new Staff {
\tempo 4 = 80
a'4 c'4 d'4 e'4
  } }
```

output.ly code

Thank You!