SMURF Programming Language Final Report

Richard Townsend, Lianne Lairmore, Lindsay Neubauer, Van Bui, Kuangya Zhai
{rt2515, lel2143, lan2135, vb2363, kz2219}@columbia.edu

December 30, 2013

1 Introduction

SMURF is a functional language that allows a composer to create serialist music based on the twelve tone composition technique. In general, serialism is a musical composition method where a set of values, chosen through some methodical progress, generates a sequence of musical elements. SMURF is based on the functional syntax and semantics set forth by Haskell. The backend of SMURF generates MIDIs corresponding to the composition defined by the user's initial program in SMURF.

1.1 Background: What is Serialism?

In general, serialism is a musical composition technique where a set of values, chosen through some methodical process, generates a sequence of musical elements. Its origins are often attributed to Arnold Schoenberg's twelve-tone technique, which he began to use in the 1920s. In this system, each note in the chromatic scale is assigned an integer value, giving us a set of twelve "pitch classes" (Figure 1 [1]). A composer utilizing this method then takes each of these integers, and orders them into a *twelve tone row*, where each number appears exactly once. We refer to this row as the *prime form* of a piece, and conventionally refer to it as P_0.

The composer can then generate other rows that are derived from P_0 through three types of transformations: transposition, inversion, and retrograde. In each of these transformations, we always use mod 12 arithmetic to preserve the numbering system of our pitch classes. Transposing a row consists of taking each pitch class in the row and adding the same number to each. If we transpose P_0 by four semitones, we add four mod twelve to each pitch class in P_0 and end up with a new row called P_4. In general, P_x is a transposition of P_0 by x semitones. To invert a row, we "flip" each interval between two pitch classes in that row. An interval is best thought of as the smallest "distance" between two pitch classes, using the proximity on the piano of the two pitch classes as the distance metric (refer to Figure 1 for reference). For example, pitch classes 0 and 11 have a distance of 1 from each other, since you can reach pitch class 0 from 11 by adding 1 to 11 (remember the mod 12 arithmetic) or reach 11 from 0 by subtracting 1 from 0. Thus an interval of +1 exists from 11 to 0, and an interval of -1 exists from 0 to 11. As a further example, if P_0 starts with pitch classes 0-11-7, then we have an interval of -1 between the first two pitches and -4 between the second two. Flipping an interval between two pitch classes is identical to negating its sign. Thus, in the inverse of P_0 (called I_0), the first interval would be +1 and the second would be +4, giving us 0-1-5 as our first three pitch classes. The subscript of I_x refers both to the number of transpositions required to arrive at I_x from I_0, and to the prime row P_x that would need to be inverted to generate I_x. The final row operation is a retrograde transformation, which merely consists of reading a row backwards. That is, R_x is generated by reading the pitch classes of P_x in their opposite order. One can also have a retrograde inversion; RI_x is generated by reading the pitch classes of I_x backwards.

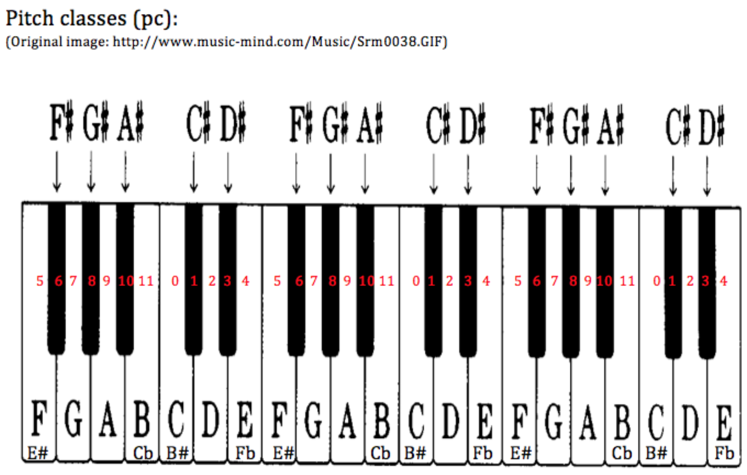
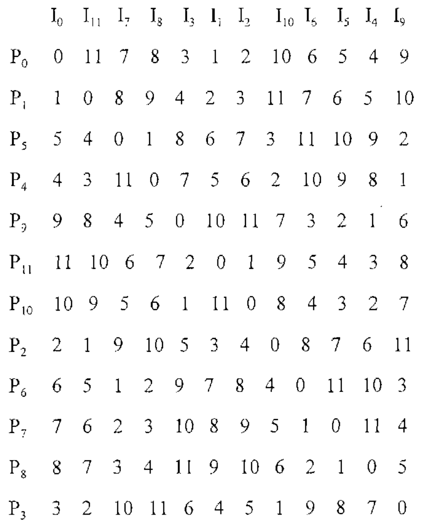

Pitch classes (pc):
(Original image: http://www.music-mind.com/Music/Srm0038.GIF)
F♯ G♯ A♯ C♯ D♯ F♯ G♯ A♯ C♯ D♯ F♯ G♯ A♯ C♯ D♯
5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4 5 6 7 8 9 10 11 0 1 2 3 4
F G A B C D E F G A B C D E F G A B C D E
E# Cb B# Fb E# Cb B# Fb E# Cb B# Fb

Figure 1: pitch classes



	I_0	I_{11}	I_7	I_8	I_3	I_1	I_2	I_{10}	I_6	I_5	I_4	I_9
P_0	0	11	7	8	3	1	2	10	6	5	4	9
P_1	1	0	8	9	4	2	3	11	7	6	5	10
P_5	5	4	0	1	8	6	7	3	11	10	9	2
P_4	4	3	11	0	7	5	6	2	10	9	8	1
P_9	9	8	4	5	0	10	11	7	3	2	1	6
P_{11}	11	10	6	7	2	0	1	9	5	4	3	8
P_{10}	10	9	5	6	1	11	0	8	4	3	2	7
P_2	2	1	9	10	5	3	4	0	8	7	6	11
P_6	6	5	1	2	9	7	8	4	0	11	10	3
P_7	7	6	2	3	10	8	9	5	1	0	11	4
P_8	8	7	3	4	11	9	10	6	2	1	0	5
P_3	3	2	10	11	6	4	5	1	9	8	7	0

Figure 2: twelve tone matrix

Once a composer chooses a P_0, the three transformations outlined above can be applied to varying degrees to generate a *twelve tone matrix*, which will contain each P row as a row in the matrix and each I row as a column. Furthermore, all of the R and RI rows are found by reading the rows in the matrix from right to left or the columns from bottom to top, respectively. An example of a twelve tone matrix from one of Shoenberg's pieces can be found in Figure 2 [2]. Finally, using the twelve tone matrix as a guide, the composer picks various rows and columns to serve as melodic and harmonic elements in their composition, resulting in a piece of serial music.

1.2 Motivation

Twelve tone serialism is a mathematically intensive method of creating music which involves mapping notes to numbers. It is natural to work with twelve tone rows using a programming language since the method treats notes like numbers that can be added and subtracted. SMURF makes twelve tone composition easier by using data types and programming paradigms that cater to the needs of a serial composer. By simplifying the method of inverting, retrograding, and transposing rows, composers can focus more on how to exploit new ways to make serial music and worry less about creating matrices.

We chose to implement a functional language because of the clear and succinct programs that functional languages produce. In addition, the well known ability of functional languages to work on lists is advantagous for twelve tone serialism, because most serial arithmetic operations use rows and columns from the twelve tone matrix as operands. As a group we were also interested on how a functional language compiler works.

Overall we hope to use the simplicity of a functional language to help composers write complex, new, and interesting music based on twelve tone serialism.

2 Tutorial

This tutorial covers how to install, run, and write basic SMURF programs.

2.1 Installation

First, untar the SMURF tarball. To compile SMURF, simply type `make` in the top level source directory. A few sample SMURF programs are located in the **examples** directory as a reference.

2.2 Compiling and Running a SMURF Program

A SMURF program has the extension **.sm**. To compile and run a SMURF program, execute the `toplevel.native` file as follows:

```
$ ./toplevel.native foo.sm
```

A midi file containing the composition defined in your SMURF program will generate if compilation was successful. The midi file can be played using any midi compatible software such as QuickTime. Running `toplevel.native` with the -h flag will display additional options that can be supplied to `toplevel.native` when compiling a SMURF program, such as specifying an output midi file name.

2.3 SMURF Examples

A basic SMURF program can generate a midi file that plays a note. The following SMURF program defines a quarter note in middle C:

```
 1  /* A quarter note in middle C - Hello World! */

    main = (0,2)$4
```

simplenote.sm

The identifier `main` must be set in every SMURF program. In simplenote.sm, main is set to a note. A note in SMURF consists of a pitch class or rest, the register, and the beat. In simplenote.sm, the pitch class is set to 0, the register is 2, and the 4 indicates a single beat, which turns the note into a quarter note.

As a second example, consider the following program that plays an ascending scale followed by a series of notes interleaved with rests:

```
    /* Sample SMURF program that plays a shortened cascade */
 2
    //[Register] -> [Pitch classses] -> [Durations] -> [Chords]
    makeChords :: [Int] -> [Int] -> [Beat] -> [Chord]
    makeChords [] _ _ = []
    makeChords _ [] _ = []
 7  makeChords _ _ [] = []
    makeChords r:restr p:restp d:restd = [(p,r)$d] : (makeChords restr restp restd)

    endBeats = [4,4,4,4,4,2]
    endReg = [0,2,2,0,2,0,2,0,2]
12  reg3 = 0 : endReg

    track1 = let pitches1 = [0,2,4,5,7,9,11,0,-1,0,-1,11,-1,11]
                 reg1 = [2,2,2,2,2,2,2,3,0,3,0,2,0,2]
                 beats1 = [8,8,8,8,8,8,8,(1 $+ 8)] ++ endBeats
17            in makeChords reg1 pitches1 beats1

    track2 = let pitches2 = [-1,11,9,-1,8,-1,8,-1,7]
                 reg2 = endReg
                 beats2 = [1,8,(2..)] ++ endBeats
22           in makeChords reg2 pitches2 beats2

    main = [track1,track2]
```

shortcascade.sm

In shortcascade.sm, `main` is set to a list of lists of chords, the latter being defined as a system in SMURF. The `makeChords` function has as input two lists of integers and a list of beats and iterates through the respective lists using recursion to generate a list of chords. The `:` operator seen in line 8 constructs a new list by appending the single note list on the left side of the operator to the list of chords. As previously mentioned, a system is a list of chords, hence `makeChords` creates a system. In line 14, a `let` expression is used to call `makeChords` providing as input the list of pitches, beats, and registers, which are defined in the declaration section of the `let` expression. Line 16 uses the concatenate operator `++` to combine two lists. On the same line, the `$+` operator performs rhythmic addition adding together a whole note and an eighth note. The `.` operator shown in line 21 also performs rythmic addition, but adds a half of the note on the left side of the operator. In this case, the dot operator adds a quarter note and an eighth note to the half note. This SMURF example introduces several SMURF language features, but there are additional features that are not shown in this example.

The remainder of this document describes in more detail the SMURF language.

3 Langauge Reference Manual

3.1 Syntax Notation

The syntax notation used in this manual is as follows. Syntactic categories are indicated by *italic* type. Literal words and characters used in the SMURF language will be displayed in `typeset`. Alternatives are listed on separate lines.

Regular expression notations are used to specify grammar patterns in this manual. $r*$ means the pattern r may appear zero or more times, $r+$ means r may appear one or more times, and $r?$ means r may appear once or not at all. *r1*|*r2* denotes an option between two patterns, and *r1 r2* denotes *r1* followed by *r2*.

3.2 Lexical Conventions

SMURF programs are lexically composed of three elements: comments, tokens, and whitespace.

3.2.1 Comments

SMURF allows nested, multiline comments in addition to single line comments.

Comment Symbols	Description	Example
`/* */`	Multiline comments, nesting allowed	`/* This /* is all */ commented */`
`//`	Single-line comment	`// This is a comment`

3.2.2 Tokens

In SMURF, a token is a string of one or more characters that is significant as a group. SMURF has 6 kinds of tokens: *identifiers*, *keywords*, *constants*, *operators*, *separators* and *newlines*.

Identifiers An identifier consists of a letter followed by other letters, digits and underscores. The letters are the ASCII characters `a-z` and `A-Z`. Digits are ASCII characters `0-9`. SMURF is case sensitive.

letter → [‘`a`’-‘`z`’ ‘`A`’-‘`Z`’]

digit → [‘`0`’-‘`9`’]

underscore → ‘`_`’

identifier → *letter* (*letter* | *digit* | *underscore*)*

Keywords Keywords in SMURF are identifiers reserved by the language. Thus, they are not available for re-definition or overloading by users.

Keywords	Descriptions
`Bool`	Boolean data type
`Int`	Integer data type
`Note`	Atomic musical data type
`Beat`	Note duration data type
`Chord`	Data type equivalent to `[Note]` type
`System`	Data type equivalent to `[Chord]` type
`let, in`	Allow local bindings in expressions
`if, then, else`	Specify conditional expression, else compulsory
`main`	Specify the value of a SMURF program

Constants In SMURF, constants are expressions with a fixed value. Integer literals and Boolean keywords are the constants of SMURF.

digit → [‘0’-‘9’]

constant → -? [‘1’-‘9’] *digit**
digit+
`True`
`False`

Operators SMURF permits arithmetic, comparison, boolean, list, declaration, and row operations, all of which are carried out through the use of specific operators. The syntax and semantics of all of these operators are described in sections 3.4.6, 3.4.7, and 3.4.8, except for declaration operators, which are described in section 3.5.

Newlines SMURF uses newlines to signify the end of a declaration, except when included in a comment or preceded by the \ token. In the latter case, the newline is ignored by the compiler (see example below). If no such token precedes a newline, then the compiler will treat the newline as a token being used to terminate a declaration.

Separators

separator → `,`
`&`
`\`

Separators in SMURF are special tokens used to separate other tokens. Commas are used to separate elements in a list. The `&` symbol can be used in place of a newline. That is, the compiler will replace all `&` characters with newlines. The \ token, when followed by a newline token, may be used to splice two lines. E.g.

```
genAltChords (x:y:ys) = [(x,Time 4,1)]    \
                        :[(y,Time 4,-1)]:(genAltChords ys)
```

is the same as

```
genAltChords (x:y:ys) = [(x,Time 4,1)]:[(y,Time 4,-1)]:(genAltChords ys)
```

The `&` and \ tokens are syntactic sugar and exist solely for code formatting when writing a SMURF program.

3.2.3 Whitespace

Whitespace consists of any sequence of *blank* and *tab* characters. Whitespace is used to separate tokens and format programs. All whitespace is ignored by the SMURF compiler. As a result, indentations are not significant in SMURF.

3.3 Meaning of Identifiers

In SMURF, an identifier is either a keyword or a name for a variable or a function. The naming rules for identifiers are defined in section 3.2.2. This section outlines the use and possible types of non-keyword identifiers.

3.3.1 Purpose

Functions Functions in SMURF enable users to structure programs in a more modular way. Each function takes at least one argument and returns exactly one value (except the built in `random` function, see section 3.6 for more details), whose types need to be explicitly defined by the programmer. The function describes how to produce the return value, given a certain set of arguments. SMURF is a side effect free language, which means that if provided with the same arguments, a function is guaranteed to return the same value (again, this is no longer the case when using the `random` function).

Variables In SMURF, a variable is an identifier that is bound to a constant value or to an expression. Any use of a variable within the scope of its definition refers to the value or expression to which the variable was bound. Each variable has a static type which can be automatically deduced by the SMURF compiler, or explicitly defined by users. The variables in SMURF are immutable.

3.3.2 Scope and Lifetime

The lexical scope of a top-level binding in a SMURF program is the whole program itself. As a result of this fact, a top-level binding can refer to any other top-level variable or function on its right-hand side, regardless of which bindings occur first in the program. Local bindings may also occur with the `let` *declarations* `in` *expression* construct, and the scope of a binding in *declarations* is *expression* and the right hand side of any other bindings in *declarations*. A variable or function is only visible within its scope. An identifier becomes invalid after the ending of its scope. E.g.

```
prime = [2,0,4,6,8,10,1,3,5,7,9,11]
main = let prime = [0,2,4,6,8,10,1,3,5,7,9,11]
           p3 = (head prime) + 3
       in (p3, 0)$4
```

In line 1, `prime` is bound to a list of integers in a top-level definition, so it has global scope. In line 2, the `main` identifier (a special keyword described in 3.5.4) is bound to a `let` expression. The `let` expression declares two local variables, `prime` and `p3`. In line 3, the `head` function looks for a definition of prime in the closest scope, and thus uses the binding in line 2. So the result of the expression in line 4 is `(3,0)$4`. After line 4 and prior to line 2, the locally defined `prime` and `p3` variables are invalid and can't be accessed.

3.3.3 Basic Types

There are three fundamental types in SMURF: `Int`, `Bool` and `Beat`.

- `Int`: integer type
- `Bool`: boolean type
- `Beat`: beat type, used to represent the duration of a note. A constant of type `Beat` is any power of 2 ranging from 1 to 16. These beat constants are assumed to be of type `Int` until they are used in an operation that requires them to have type `Beat` e.g. when used as an operand to the beat arithmetic operator `$+`.

3.3.4 Structured Types

Structured types use special syntactic constructs and other types to describe new types. There are two structured types in SMURF: *list* types and *function* types.

A *list* type has the format `[`t`]` where t is a type that specifies the type of all elements of the list. Thus, all elements of a list of type `[`t`]` must themselves have type t. Note that t itself may be a list type.

A *function* type has the format t_1 `->` t_2 `->` ... `->` t_n `->` t_{ret} which specifies a function type that takes n arguments, where the kth argument has type t_k, and returns an expression of type t_{ret}. Any type may be used to define a function type, except for a function type itself. In other words, functions may not be passed as arguments to other functions, nor may a function return another function.

3.3.5 Derived Types

Besides the basic types, SMURF also has several derived types.

Expressions of type `Note` are used to represent musical notes in SMURF. The note type can be written as

(*Int*, *Int*)`$`*Beat*`[.]*`

The first expression of type `Int` must evaluate to an integer in the range from -1 to 11, representing a pitch class or a rest. When this expression evaluates to -1, the note is treated as a rest, otherwise it represents the pitch class of the note. The second expression of type `Int` must evaluate to an integer in the range of 0-3, representing the register of the note, where the integer values and corresponding registers are given below.

- 1: Bass clef, B directly below middle C to first C below middle C
- 0: Bass clef, next lowest B to next lowest C
- 2: Treble clef, middle C to the first B above middle C
- 3: Treble clef, first C above middle C to next highest B

The expression of type `Beat` refers to the duration of the note, and may be followed by optional dots. The dot is a postfix operator described in section 3.4.6. Ignoring the possible postfix operators, the expression must evaluate to an integer in the range [1,2,4,8,16]. Using this format, a quarter note on middle C could be written as `(0,2)$4`.

The `Chord` type is used to represent several notes to be played simultaneously. It is equivalent to the list type `[Note]`. The compiler will check to make sure all the notes in a chord have the same time duration.

The `System` type is used to represent a list of chords to be played sequentially. It is equivalent to the list type `[Chord]`.

3.3.6 Polymorphic Types

SMURF provides the option of specifying an identifier as having a polymorphic type by using a non-keyword identifier in place of a basic, structured, or derived type in that identifier's type signature. For more information on the structure of type signatures, see section 3.5.1. For example, `a :: b` specifies that a variable named `a` has polymorphic type `b`, where `b` can be replaced with any basic, structured, or derived type. Using the same polymorphic type across different type signatures is permitted and each use has no bearing on another. For example, giving `a :: b` and `c :: b` merely states that `a` and `c` are both variables with polymorphic types and would be equivalent to giving `a :: hippo` and `c :: dinosaur`. However, if the same identifier is used multiple times as a type in a function's type signature, then the types assigned to those components of the function must be identical. For example, say we have a function

```
f :: Int -> b -> [b]
```

This type signature specifies that `f` takes two arguments, the first of type `Int` and the second of polymorphic type, and that the expression bound to `f` must be a list type, where the elements of the list are of the same type as the second argument passed to `f`. Thus `f 0 True = [False]` would be a valid function declaration (as `True` and `False` both have type `Bool`) given this type signature, but `f 0 True = [1]` would result in a compile-time error because `1` has type `Int`.

3.4 Expressions

This section describes the syntax and semantics of *expressions* in SMURF. Some expressions in SMURF use prefix, infix, or postfix operators. Unless otherwise stated, all infix and postfix operators are left-associative and all prefix operators are right-associative. Some examples of association are given below.

Expression	Association
`f x + g y - h z`	`((f x) + (g y)) - (h z)`
`let { ... } in x + y`	`let { ... } in (x + y)`
`~ <> [0,1,2,3,4,5,6,7,8,9,10,11]`	`(~ (<> [0,1,2,3,4,5,6,7,8,9,10,11]))`

3.4.1 Variable Expression

variable-expr → *variable*

variable → *identifier*

A variable x is an expression whose type is the same as the type of x. When we evaluate a variable, we are actually evaluating the expression bound to the variable in the closest lexical scope. A variable is represented with an *identifier* as defined in section 3.2.2.

3.4.2 Constant Expression

constant-expr → *constant*

An integer or boolean constant, as described in section 3.2.2, is an expression with type equivalent to the type of the constant.

3.4.3 Parenthesized Expression

parenthesized-expr → `(` *expression* `)`

An expression surrounded by parentheses is itself an expression. Parentheses can be used to force the evaluation of an expression before another e.g. `2 + 3 - 4 - 5` evaluates to `((2+3) - 4) - 5 = -4`
but `2 + 3 - (4 - 5)` evaluates to `(2 + 3) - (4 - 5) = 6`.

3.4.4 List Expression

list-expr → `[ ]`
`[` *expression* (`,` *expression*)* `]`

A list is an expression. Lists can be written as:

`[`*expression*$_1$`,` ...`,` *expression*$_k$`]`

or

expression$_1$`:`*expression*$_2$`:` ... `:`*expression*$_k$`:[]`

where $k >= 0$. These two lists are equivalent. The expressions in a list must all be of the same type. The empty list `[]` has a polymorphic type i.e. it can take on the type of any other list type depending on the context.

3.4.5 Notes

note-expr → `(`*expression*`,` *expression*`)$`*expression*

A note is an expression, and is written as a tuple of expressions of type `Int` followed by a `$` symbol and an expression of type `Beat`. The values of each of these expressions must follow the rules outlined in section 3.3.5.

3.4.6 Postfix Operator Expressions

postfix-expression → *expression*`.`

The only expression in SMURF using a postfix operator is the partial augmentation of an expression of type `Beat`, which uses the dot operator. This operator has higher precedence than any prefix or infix operator. We say "partial augmentation" because a dot increases the durational value of the expression to which it is applied, but only by half of the durational value of that expression. That is, if *expr* is an expression of type `Beat` that evaluates to a duration of n, then *expr*`.` is a postfix expression of type `Beat` that evaluates to a duration of $n + n/2$. In general, a note with duration d and total dots n has a total duration of $2d - d/2^n$. The dot operator may be applied until it represents an addition of a sixteenth note duration, after which no more dots may be applied. For instance, `4..` is legal, as this is equivalent to a quarter note duration plus an eighth note duration (the first dot) plus a sixteenth note duration (the second dot). However, `8..` is not legal, as the second dot implies that a thirty-second note duration should be added to the total duration of this expression. Our compiler checks the number of dots and returns an error if too many are applied.

3.4.7 Prefix Operator Expressions

prefix-expression → *prefix-op expression*

Prefix Operator	Description	Example
∼	Tone row inversion	∼ `row` (returns the inversion of `row`)
`<>`	Tone row retrograde	`<> row` (returns the retrograde of `row`)
`!`	Logical negation	`if !(a == 5) then True else False`

SMURF has three prefix operators: logical negation, tone row inversion, and tone row retrograde. There is another row transformation operator, but it takes multiple arguments and is described in section 3.4.8. The tone row prefix operators have higher precedence than any infix operator, while the logical negation operator is lower in precedence than all infix operators except for the other logical operators `&&` and ||. The logical negation operator can only be applied to expressions of type `Bool`, and the two row operators can only be applied to expressions of type `[Int]`. The compiler will check that all of the integers in a list are in the range $0 - 11$ if the list is passed to either of the tone row operators. All three operators return an expression of the same type as the expression the operator was applied to.

3.4.8 Binary Operator Expressions

binary-expression → *expression*$_1$ *binary-op expression*$_2$

The following categories of binary operators exist in SMURF, and are listed in order of decreasing precedence: list, arithmetic, comparison, boolean, tone row.

List Operator	Description	Example
`++`	List Concatenation	`[1,2,3] ++ [4,5,6]` (result is `[1,2,3,4,5,6]`)
`:`	List Construction	`1 : [2,3,4]` (result is `[1,2,3,4]`)

List operators List operators are used to construct and concatenate lists. These two operators are `:` and `++`, respectively. The `:` operator has higher precedence than the `++` operator. Both of these operators are right-associative. The list construction operator requires that $expression_2$ be an expression of type `[`t`]`, where t is the type of $expression_1$. In other words, $expression_1$ must have the same type as the other elements in $expression_2$ when doing list construction. When doing list concatenation, both $expression_1$ and $expression_2$ must have type `[`t`]`, where t is some non-function type.

Arithmetic Operator	Description	Example
`+`	Integer Addition	`a + 2`
`-`	Integer Subtraction	`5 - a`
`*`	Integer Multiplication	`5 * 10`
`/`	Integer Division	`4 / 2`
`%`	Integer Modulus, ignores negatives	`14 % 12`
`%+`	Pitch Class Addition (addition mod 12)	`14 %+ 2 == 4`
`%-`	Pitch Class Subtraction (subtraction mod 12)	`14 %- 2 == 0`
`$+`	Rhythmic Addition	`2 $+ 2 == 1`
`$-`	Rhythmic Subtraction	`1 $- 2 == 2`
`$*`	Rhythmic Augmentation	`8 $* 4 == 2`
`$/`	Rhythmic Diminution	`2 $/ 8 == 16`

Arithmetic operators There are three types of arithmetic operators: basic, pitch class, and rhythmic. Basic arithmetic operators are those found in most programming languages like `+`, `-`, `*`, `/`, and `%`, which operate on expressions of type `Int`. It should be noted that the modulus operator ignores negatives e.g. `13 % 12` is equal to `-13 % 12` is equal to `1`. The pitch class operators are `%+` and `%-`. These can be read as mod 12 addition and mod 12 subtraction. They operate on expressions of type `Int`, but the expressions must evaluate to values in the range $0 - 11$. The built-in mod 12 arithmetic serves for easy manipulation of pitch class integers. Lastly, there are rhythmic arithmetic operators (both operands must be of type `Beat`). These include `$+`, `$-`, `$*`, and `$/`. If one of the operands of these operators is of type `Int`, it will be cast to a `Beat` type if it is an allowable power of 2 and generate a semantic error otherwise.

In terms of precedence, `*`, `/`, `$*`, `$/` and `%` are all at the same level of precedence, which is higher than the level of precedence shared by the rest of the arithmetic operators.

Comparison Operator	Description	Example
`<`	Integer Less than	`if a < 5 then True else False`
`>`	Integer Greater than	`if a > 5 then True else False`
`<=`	Integer Less than or equal to	`if a <= 5 then True else False`
`>=`	Integer Greater than or equal to	`if a >= 5 then True else False`
`$<`	Rhythmic Less than	`4 $< 8 == False`
`$>`	Rhythmic Greater than	`4 $> 8 == True`
`$<=`	Rhythmic Less than or equal to	`4 $<= 4 == True`
`$>=`	Rhythmic Greater than or equal to	`1 $>= 16 == True`
`==`	Structural comparison	`if a == 5 then a = True else a = False`

Comparison operators SMURF allows comparison operations between expressions of type `Int` or `Beat`. Structural comparison, however, can be used to compare expressions of any type for equality. All of the comparison operators have the same precedence except for structural comparison, which has lower precedence than all of the other comparison operators.

Boolean Operator	Description	Example
`&&`	Logical conjunction	`if b && c then True else False`
`||`	Logical disjunction	`if b || c then True else False`

Boolean operators Boolean operators are used to do boolean logic on expressions of type `Bool`. Logical conjunction has higher precedence than logical disjunction.

Tone row operators The only binary tone row operator is the transposition operator, $\wedge\wedge$. $expression_1$ must have type `Int`, and $expression_2$ must be an expression that evaluates to a list of pitch classes. The result of this operation is a new tone row where each pitch class has been transposed up by n semitones, where n is the result of evaluating $expression_2$.

3.4.9 Conditional expressions

conditional-expression $\rightarrow$ `if` $expression_{boolean}$ `then` $expression_{true}$ `else` $expression_{false}$

When the value of $expression_{boolean}$ evaluates to true, $expression_{true}$ is evaluated, otherwise $expression_{false}$ is evaluated. $expression_{boolean}$ must have type `Bool`.

3.4.10 Let Expressions

let-exp $\rightarrow$ `let` *decls*+ `in` *expression*

Let expressions have the form `let` *decls* `in` *e*, where *decls* is a list of one or more declarations and *e* is an expression. The scope of these declarations is discussed in section 3.5.

The declarations in a let expression must be separated by either the `&` symbol or by the newline character. For example:

```
let x = 2 & y = 4 & z = 8
in x + y + z
```

The previous code is equivalent to the following:

```
let x = 2
    y = 4
    z = 8
in x + y + z
```

If the first code snippet were written without the & symbol and no newlines after each declaration, a compile-time error will be raised.

3.4.11 Function application expressions

function-app-expression $\rightarrow$ *identifier expression*+

A function gets called by invoking its name and supplying any necessary arguments. Functions can only be called if they have been declared in the same scope where the call occurs, or in a higher scope. Functions may be called recursively. Function application associates from left to right. Parentheses can be used to change the precedence from the default. Furthermore, parentheses must be used when passing the result of a complex expression to a function. Here, complex expression refers to any expression that uses an operator or itself is a function call. The following evaluates function *funct1* with argument *b* then evaluates function *funct2* with argument *a* and the result from evaluating (*funct1 b*):

funct2 a (funct1 b)

If the parentheses were not included, a compile-time error would be generated, as it would imply that *funct2* would be called with *a* as its first argument and *funct1* as its second argument, which is illegal based on the description of function types in section 3.3.4.

A function call may be used in the right-hand side of a binding just like any other expression. For example:

```
let a = double 10
in a
```

evaluates to 20, where `double` is a function that takes a single integer argument and returns that integer multiplied by two.

3.5 Declarations and Bindings

This section of the LRM describes the syntax and informal semantics of declarations in SMURF. A program in SMURF, at its top-most level, is a series of declarations separated by newline tokens. Declarations may also occur inside of `let` expressions (but still must be separated with newline tokens). The scoping of such declarations is described in this section. There are three types of declarations in SMURF: type signatures, definitions, and function declarations.

Declaration Operator	Description	Example
`::`	Type specification	`number :: Int`
`->`	Argument and function return type specification	`isPositiveNum :: Int -> Bool`
`=`	Variable or function binding	`x = 3`

3.5.1 Type Signatures

type-sig $\rightarrow$ *identifier* `::` (*type*|*function-type*)

function-type $\rightarrow$ *type* `->` *type* (`->` *type*)*

type → `Int`
`Bool`
`Beat`
`Note`
`Chord`
`System`
identifier
`[` *type* `]`

A type signature explicitly defines the type for a given identifier. The `::` operator can be read as "has type of." Only one type signature for a given identifier can exist in a given scope. That is, two different type signatures for a given identifier can exist, but they must be declared in different scopes. There are four categories of types in SMURF: basic, structured, derived, and polymorphic types; types are described in sections 3.3.3-3.3.6.

3.5.2 Definitions

definition → *identifier* = *expression*

A definition binds an identifier to an expression. All definitions at a given scope must be unique and can be mutually recursive. For example, the following is legal in SMURF:

```
let x = 4
   z = if y == 7 then x else y
   y = let x = 5
        in x + 3
in x + z + y
```

The x in the nested let expression is in a different scope than the x in the global let expression, so the two definitions do not conflict. z is able to refer to y even though y is defined after z in the program. In this example, the three identifiers x, y, and z in the global `let` will evaluate to values 4, 8, and 8, respectively, while the identifier x in the nested let expression will evaluate to 5.

A type signature may be given for the identifier in a definition but is not required.

3.5.3 Function Declarations

fun-dec → *identifier args* = *expression*

args → *pattern*
pattern args

pattern → *pat*
pat `:` *pattern*
`[` *pattern-list*? `]`
`(` *pattern* `)`

pattern-list → *pat* `(,` *pat*`)`*

pat → *identifier*
constant
`_`

A function declaration defines an identifier as a function that takes some number of expressions as arguments and, based on which patterns are matched against those expressions when the function is called, returns the result of a given expression. Essentially, a function declaration can be seen as a binding associating an expression with a function identifier and a set of patterns that will be matched against the function's arguments when the function is called. There must be at least one pattern listed as an argument in a function declaration. All function declarations for the same identifier in a given scope must have the same number of patterns given in the declaration.

Unlike variable definitions, multiple function declarations for the same identifier may exist in the same scope, as long as no two declarations have an equivalent set of patterns. This rule does not pertain to multiple function declarations for an identifier across different scopes.

If a function declaration for some identifier x occurs in scope n, then a type signature for x in scope $k >= n$ is required. That is if a function has been declared but its type has not been explicitly stated in the same or a higher scope, a compile-time error will be generated. The type of the patterns in a function declaration are checked at compile-time as well, and an error is issued if they don't match the types specified in that function's type signature.

A *pattern* can be used in a function declaration to "match" against arguments passed to the function. The arguments are evaluated and the resultant values are matched against the patterns in the same order they were given to the function. If the pattern is a constant, the argument must be the same constant or evaluate to that constant value in order for a match to occur. If the pattern is an identifier, the argument's value is bound to that identifier in the scope of the function declaration where the pattern was used. If the pattern is the wildcard character '`_`', any argument will be matched and no binding will occur. If the pattern is structured, the argument must follow the same structure in order for a match to occur.

Below, we have defined an example function `f` that takes two arguments. The value of the function call is dependent on which patterns are matched. The most restrictive patterns are checked against the arguments first. In this example, we first check if the first argument evaluates to `0` (we disregard the second argument using the wildcard character), and return `True` if it does. Otherwise, we check if the second argument evaluates to the empty list, and, if so, return `False`. Next, we check if the second argument evaluates to a list containing exactly two elements and, if so, the first element is bound to `x` and the second is bound to `y` in the expression to the right of the binding operator `=`, and that expression is evaluated and returned. Finally, if none of the previous pattern sets matched, we bind the first argument to `m`, the head of the second argument to `x`, and the rest of the second argument to `rest`. Note we can do this as we already checked if the second argument was the empty list, and, since we did not match that pattern, we can assume there is at least one element in the list.

```
f :: Int -> [Int] -> Bool
f _ [] = False
f _ [x, y] = if x then True else False
f 0 _ = True
f m x:rest = f m rest
```

3.5.4 `main` Declaration

Every SMURF program must provide a definition for the reserved identifier `main`. This identifier may only be used on the left-hand side of a top-level definition. The expression bound to `main` is evaluated and its value is the value of the SMURF program itself. That is, when a SMURF program is compiled and run, the expression bound to `main` is evaluated and the result is converted to our bytecode representation of a MIDI file. As a result, this expression must evaluate to a value of type `[]`, `Note`, `Chord`, `System`, or `[System]` (or any type that is equivalent to one of these). If a definition for `main` is not included in a SMURF program or if the expression bound to it does not have one of the types outlined above, a compile-time error will occur.

3.6 Library Functions

Below are the library functions that can be used in the SMURF language. While some of these functions are implemented using SMURF, others (such as print) are special cases that provide helpful services but cannot explicitly be defined in our language. These special cases are implemented in the translation section of the compielr. Each library function will include a description and its SMURF definition (if it can be defined using SMURF). Users are not permitted to redefine any of these functions.

Print

The function `print` takes an argument of any type, evaluates it, and prints the result to standard output. The result of calling the `print` function is the result of evaluating its argument i.e. `print(x+1)` evaluates to `x+1`.

Random

The function `random` is the only SMURF function that takes no parameters. It is another example of a function that cannot be explicitly defined using the SMURF language. The result of a call to `random` is a pseudo-random integer between 1 and 1000000, inclusive. For example, `random % 12` will return some number between 0 and 11. Every time `random` is called, a new pseudo-random seed is used to initialize the random number generator used by the compiler, allowing for different results on each run of a program where random is used. There is no capability for the user to set their own initializing seed.

Head

The function `head` takes a list as an argument and returns the first element. This function is commonly used when working with lists.

```
head :: [a] -> a
head (h:tl) = h
```

Tail

The function `tail` takes a list as an argument and returns the same list with the first element removed. This function is commonly used when working with lists.

```
tail :: [a] -> [a]
tail (h:tl) = tl
```

MakeNotes

The function `makeNotes` takes in three lists and returns a list of notes. The first list consists of expressions of type `Int` representing pitches and/or rests. The second list consists of expressions of type `Int` representing the register that the pitch will be played in. The third list is a list of expressions of type `Beat` representing a set of durations. This function allows the user to manipulate tone rows independently of beats and registers, then combine the three components into a list of notes. If the lengths of the three arguments are not equivalent, this function will only return the list of notes generated from the first n elements of each list, where n is the length of the shortest list given as an argument.

```
makeNotes :: [Int] -> [Int] -> [Beat] -> [Note]
makeNotes [] _ _ = []
makeNotes _ [] _ = []
makeNotes _ _ [] = []
makeNotes (h1:tl1) (h2:tl2) (h3:tl3) = (h1,h2)$h3:(makeNotes tl1 tl2 tl3)
```

Reverse

The function `reverse` takes a list as an argument and returns the same list in reverse.

```
reverse :: [a] -> [a]
reverse [] = []
reverse a:rest = (reverse rest) ++ [a]
```

Last

The function `last` takes a list as an argument and returns the last element in the list.

```
last :: [a] -> a
last a:[] = a
last a:rest = last rest
```

Drop

The function `drop` takes an integer n and a list as arguments, and returns the same list with the first n elements removed.

```
drop :: Int -> [a] -> [a]
drop 0 x = x
drop _ [] = []
drop x l:rest = drop (x - 1) rest
```

Take

The function `take` takes an integer n and a list as arguments, and returns a list composed of the first n elements of the original list.

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take x l:rest = l : (take (x - 1) rest)
```

4 Project Plan

4.1 Example Programs

The first sample program constructs a little tune. First an ascending scale is heard, followed by a descending scale being played in four tracks, with each track suspending the second note it plays in the descending scale. Finally, we hear a half-diminished chord, a fully diminished chord, and a major seventh chord, with the chords being interleaved with quarter rests.

```
/* Sample SMURF program that should play a cascade :-) */

//[Register] -> [Pitch classses] -> [Durations] -> [Chords]
makeChords :: [Int] -> [Int] -> [Beat] -> [Chord]
makeChords [] _ _ = []
makeChords _ [] _ = []
makeChords _ _ [] = []
makeChords r:restr p:restp d:restd = [(p,r)$d] : (makeChords restr restp restd)

endBeats = [4,4,4,4,4,2]
endReg = [0,2,2,0,2,0,2,0,2]
reg3 = 0 : endReg

track1 = let pitches1 = [0,2,4,5,7,9,11,0,-1,0,-1,11,-1,11]
             reg1 = [2,2,2,2,2,2,2,3,0,3,0,2,0,2]
             beats1 = [8,8,8,8,8,8,8,(1 $+ 8)] ++ endBeats
          in makeChords reg1 pitches1 beats1

track2 = let pitches2 = [-1,11,9,-1,8,-1,8,-1,7]
             reg2 = endReg
             beats2 = [1,8,(2..)] ++ endBeats
         in makeChords reg2 pitches2 beats2


track3 = let pitches3 = [-1,-1,7,5,-1,5,-1,5,-1,4]
             beats3 = [1,4,8,(2 $+ 8)] ++ endBeats
         in makeChords reg3 pitches3 beats3

track4 = let pitches4 = [-1,-1,4,2,-1,2,-1,2,-1,0]
             beats4 = [1,2,8,4.] ++ endBeats
             reg4 = reg3
         in makeChords reg4 pitches4 beats4

main = [track1,track2,track3,track4]
```

cascade.sm

```

Timing Resolution set to 4 PPQ

Instrument set to 48 on channel 0
Instrument set to 48 on channel 1
Instrument set to 48 on channel 2
Instrument set to 48 on channel 3

Track 0:
 tick 0, channel 1: program change 48
 tick 0, channel 1: note C4 on velocity: 90
 tick 2, channel 1: note C4 on velocity: 0
 tick 2, channel 1: note D4 on velocity: 90
 tick 4, channel 1: note D4 on velocity: 0
 tick 4, channel 1: note E4 on velocity: 90
 tick 6, channel 1: note E4 on velocity: 0
 tick 6, channel 1: note F4 on velocity: 90
 tick 8, channel 1: note F4 on velocity: 0
 tick 8, channel 1: note G4 on velocity: 90
 tick 10, channel 1: note G4 on velocity: 0
 tick 10, channel 1: note A4 on velocity: 90
 tick 12, channel 1: note A4 on velocity: 0
 tick 12, channel 1: note B4 on velocity: 90
 tick 14, channel 1: note B4 on velocity: 0
 tick 14, channel 1: note C5 on velocity: 90
 tick 32, channel 1: note C5 on velocity: 0
 tick 36, channel 1: note C5 on velocity: 90
 tick 40, channel 1: note C5 on velocity: 0
 tick 44, channel 1: note B4 on velocity: 90
 tick 48, channel 1: note B4 on velocity: 0
 tick 52, channel 1: note B4 on velocity: 90
 tick 60, channel 1: note B4 on velocity: 0
 tick 60, end of track
Track 1:
 tick 0, channel 2: program change 48
 tick 16, channel 2: note B4 on velocity: 90
 tick 18, channel 2: note B4 on velocity: 0
 tick 18, channel 2: note A4 on velocity: 90
 tick 32, channel 2: note A4 on velocity: 0
 tick 36, channel 2: note G#4 on velocity: 90
 tick 40, channel 2: note G#4 on velocity: 0
 tick 44, channel 2: note G#4 on velocity: 90
 tick 48, channel 2: note G#4 on velocity: 0
 tick 52, channel 2: note G4 on velocity: 90
 tick 60, channel 2: note G4 on velocity: 0
 tick 60, end of track
Track 2:
 tick 0, channel 3: program change 48
 tick 20, channel 3: note G4 on velocity: 90
 tick 22, channel 3: note G4 on velocity: 0
 tick 22, channel 3: note F4 on velocity: 90
 tick 32, channel 3: note F4 on velocity: 0
 tick 36, channel 3: note F4 on velocity: 90
 tick 40, channel 3: note F4 on velocity: 0
 tick 44, channel 3: note F4 on velocity: 90
 tick 48, channel 3: note F4 on velocity: 0
 tick 52, channel 3: note E4 on velocity: 90
 tick 60, channel 3: note E4 on velocity: 0
 tick 60, end of track
Track 3:
 tick 0, channel 4: program change 48
 tick 24, channel 4: note E4 on velocity: 90
 tick 26, channel 4: note E4 on velocity: 0
 tick 26, channel 4: note D4 on velocity: 90
 tick 32, channel 4: note D4 on velocity: 0
 tick 36, channel 4: note D4 on velocity: 90
 tick 40, channel 4: note D4 on velocity: 0
 tick 44, channel 4: note D4 on velocity: 90
 tick 48, channel 4: note D4 on velocity: 0
 tick 52, channel 4: note C4 on velocity: 90
 tick 60, channel 4: note C4 on velocity: 0
 tick 60, end of track
PASSED SEMANTIC CHECKS
java -jar ./Lib/CSV2MIDI.jar a.csv a.midi
===== Program Successfully Finished =====
===== Result Writen to a.midi =====
```

cascade.out

Our second sample program constructs the first half of Webern's Op. 27 second movement, with some liberties taken with respect to the durations of the notes and the rests (we don't have grace notes in our language which feature prominently in the actual composition). This piece has a number of interesting features, focusing on symmetry throughout the piece. For example, the tone row that starts in the right hand is completed by the left hand and vice versa for the tone row starting in the left hand. Furthermore, the next two tone rows are selected by looking at the last note in the first two tone rows, and selecting rows that start with those last notes. A number of other interesting features can be found in the composition, see Solomon's analysis [3] for a fuller description and the original score.

```
getTransRow :: [Int] -> Int -> [Int]
getTransRow [] _ = []
getTransRow l 1 = ~l
getTransRow l 2 = <>l
getTransRow l 3 = <>(~l)


/*Given a P0 and a pitch class x get the row of type 'typetrans' derived from P0
whose first element is x
For typetrans=0 we get P
              1 we get I
              2 we get R
              3 we get RI */
findRowStartsWith :: [Int] -> Int -> Int -> [Int]
findRowStartsWith [] _ _ = []
findRowStartsWith l x typetrans = if x < 0 || typetrans > 12 then [] else \
              if head (checkTrans typetrans l) == x then (checkTrans typetrans l) else \
                                    (12 - ((head (checkTrans typetrans l)) - x)) ^^ (checkTrans typetrans l
     )
checkTrans :: Int -> [Int] -> [Int]
checkTrans typetrans l = if typetrans == 0 then l else getTransRow l typetrans

singleChords :: [Note] -> [Chord]
singleChords [] = []
singleChords n:rest = [n] : (singleChords rest)

//Converts single note chords from index x to y-1 into a chord composed of the notes,
//where x is 1st arg and y is 2nd arg
subChord :: Int -> Int -> [Note] -> [Chord]
subChord _ _ [] = []
subChord 0 y l  = (take y l) : (singleChords (drop y l))
subChord x y c:rest = [c] : (subChord (x - 1) (y - 1) rest)

interweaveRest :: Int -> [Chord] -> [Chord]
interweaveRest _ [] = []
interweaveRest 1 l:rest = l : [(-1,0)$8] : (interweaveRest 1 rest)
interweaveRest 2 l:rest =  [(-1,0)$8] : l :  (interweaveRest 2 rest)


swapAt :: Int -> [Int] -> [Int] -> [Int]
swapAt 0 _ b = b
swapAt _ [] b = b
swapAt _ _ [] = []
swapAt x a:ra b:rb = a : (swapAt (x-1) ra rb)


crossOver :: Int -> [Int] -> [Int] -> [[Int]]
crossOver x a b = (swapAt x a b) : ((swapAt x b a) : [])


P0 = [0,8,7,11,10,9,3,1,4,2,6,5]
RP3 = <>(3 ^^ P0)
RI3 = <>(3 ^^ ~P0)

/* First two tone rows */
firstrows = crossOver 9 RP3 RI3
 t1 = head firstrows
t2 = head (tail firstrows)
reg = [0,1,2,3,2,1,0,1,2,3,2,1]
durations1 = [8,8,4,16,16,8,8,8,8,8,8,16]
section1 = subChord 5 8 (makeNotes t1 reg durations1)
section2 = subChord 5 8 (makeNotes t2 (<>reg) durations1)

tempt3 = findRowStartsWith P0 (last t1) 2
tempt4 = findRowStartsWith P0 (last t2) 3
secrows = crossOver 5 tempt3 tempt4
t3 = tail (head secrows) /*First note of this row is last note of previous row! */
t4 = tail (head (tail secrows)) /* Same for this row */
durations2  = [16,4,16,16,8,8,8,8,8,8,8]
newsection1 = section1 ++ (subChord 4 6 (makeNotes t3 (tail reg) durations2))
newsection2 = section2 ++ (subChord 4 6 (makeNotes t4 (tail (<>reg)) durations2))

/* This should be everything before the first repeat */
firstRepeat1 = newsection1 ++ newsection1
```

```
firstRepeat2 = newsection2 ++ newsection2

/* Second part of composition */
tempt5 = findRowStartsWith P0 (last t3) 3
tempt6 = findRowStartsWith P0 (last t4) 2
thirdrows = crossOver 11 tempt5 tempt6
t5 = tail (head thirdrows)
t6 = tail (head (tail thirdrows))
durations3 = [8,8,8,4,8,8,8,8,8,8,16]
section3 = subChord 4 6 (makeNotes t5 (tail reg) durations3)
section4 = subChord 4 6 (makeNotes t6 (tail (<>reg)) durations3)

tempt7 = findRowStartsWith P0 (last t5) 3
tempt8 = findRowStartsWith P0 (last t6) 2
fourthrows = crossOver 2 tempt7 tempt8
t7 = tail (head fourthrows)
t8 = tail (head (tail fourthrows))
durations4 = [16,16,16,8,8,8,8,8,16,16,8]
newsection3 = section1 ++ (subChord 4 6 (makeNotes t7 (tail reg) durations4))
newsection4 = section2 ++ (subChord 4 6 (makeNotes t8 (tail (<>reg)) durations4))

/* This should be everything after the first repeat */
secRepeat1 = newsection3 ++ newsection3
secRepeat2 = newsection4 ++ newsection4

main = [interweaveRest 1 firstRepeat1, interweaveRest 2 firstRepeat2]
```

webern.sm

```
Fatal error: exception Sast.Function_not_defined("head")
```

weber.out

4.2 Processes

4.2.1 Planning

We had a 2 hour meeting every Wednesday that everyone attended. In these meetings, organized by Richard (our benevolent dictator), we discussed project milestones, delegated responsibilities to each member of the group, designed and updated our design of SMURF, and eventually coded in meetings to allow for discussion of tricky parts of our implementation. We chose milestones based on a review of previous semesters team projects that were successful.

4.2.2 Specification

Both our Proposal and LRM were completely outlined in our group meetings. Lindsay took notes for the group and pushed them to GitHub so all members had access. We divided the outlines into equal sections to divide the writing and proof-reading responsibilities: Each group member had a portion to write and a different portion to proofread. Once we started coding, any updates that needed to be made were done by the person coding that portion of the language (regardless of who originally wrote that section of the LRM).

4.2.3 Development

Each member of our group was given a slice of our language to implement from start to finish. By doing this, we minimized the issues that arise from having to wait for another group member's section of the code to be implemented before being able to start your own. We each followed a similar development process, implementing in the same order the scanner (first), parser, abstract syntax tree, semantic abstract syntax tree, and code generation (last). We used GitHub to track our code but did not utilize its branching features for coding. This was a decision we made to force our code to always be in a compilable/runnable form and to avoid large merging issues at the end of development. Because we divided the language into pieces and had complete ownership of our slice, using the LRM (which we worked on together) as the ultimate reference on how to implement our section was crucial. In the few cases where the LRM specification was unable to be implemented in the way we planned, the owner of that section chose how to most appropriately implement it and then updated the LRM and the rest of the group with the changes.

4.2.4 Testing

At the end of each stage of development, every group member wrote unit tests to ensure their slice of the code worked as anticipated. Our integration testing took the form of several "Hello World" style programs. Any failed tests were addressed as soon as the failure was discovered.

4.3 Style Guide

We conformed to the following coding conventions:

- Indentation: 4 spaces, with multiple indents to differentiate between nested blocks of code
- Maximum Characters per Line: 100 (including indentation)

4.4 Project Timeline

Our project timeline includes *class deadlines* and self imposed deadlines.

Date	Milestone
09-25-13	*Proposal due*
10-07-13	Initial LRM section written
10-09-13	Initial LRM section proofread
10-27-13	Full proofread of LRM completed
10-28-13	*LRM due*
10-28-13	Scanner and Parser completed
11-06-13	Scanner and Parser tests completed
11-20-13	Semantic Analyzer completed
11-27-13	Semantic Analyzer tests completed
12-04-13	End-to-end "Hello World" compilation succeeds
12-20-13	*Final report due*

4.5 Roles and Responsibilities

Team Member	Responsibilities
Van Bui	Proposal: Examples
	Rough Draft of LRM: Write Parenthetical Expressions, Let Expressions
	Rough Draft of LRM: Proofread Description of Precedence, Syntax Notation, Library Functions, Declarations and Bindings
	Code: Function Application, Let Expressions, test scripts
Lianne Lairmore	Proposal: Motivation
	Rough Draft of LRM: Write Description of Precedence, Syntax Notation, Library Functions
	Rough Draft of LRM: Proofread Lexical Conventions, Primary Expressions, Meaning of Identifiers
	Code: Literals, Main/Print/Random, Symbol Table/Environment, Polymorphism
Lindsay Neubauer	Note Taker
	Proposal: Language Description
	Rough Draft of LRM: Write Curried Applications, Operator Application, Conditionals, Lists, Tuples
	Rough Draft of LRM: Proofread Parenthetical Expressions, Let Expressions, Type Signatures, Pattern Matching
	Code: Non-music operators, Notes, Beats, Music operators
Richard Townsend	Group Leader
	Proposal: Background
	Rough Draft of LRM: Write Declarations and Bindings
	Rough Draft of LRM: Proofread Curried Applications, Operator Application, Conditionals, Lists, Tuples
	Code: Pattern Matching, Bindings, Function Application
Kuangya Zhai	GitHub and LaTeX Go-To Person
	Proposal: Generate Latex
	Rough Draft of LRM: Write Lexical Conventions, Primary Expressions, Meaning of Identifiers
	Rough Draft of LRM: Proofread Declarations and Bindings
	Code: MIDI generation, List operators, Conditionals, Function Application

4.6 Software Development Environment

We used the following tools and languages:

- Compiler Implementation: OCaml, version 4.01.0
- Musical Interface: MIDI, java package CSV2MIDI (uses java.sound.midi.*) [4]
- Testing Environment: Shell Scripts
- Version Control System: GitHub

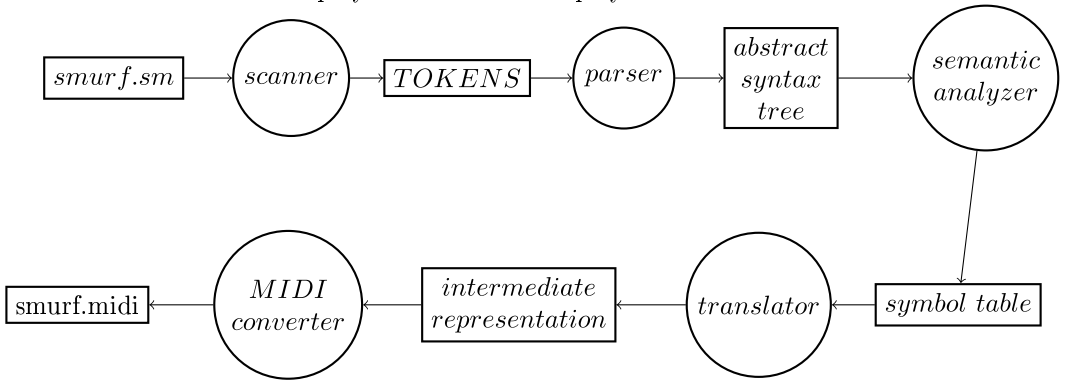
4.7 Project Log

Date	Milestone
09-18-13	Proposal writing sections assigned
	GitHub repository setup
09-25-13	LRM timeline established
10-02-13	LRM writing and proofreading sections assigned
10-11-13	Switch from OpenGL musical score to MIDI music
10-16-13	Decided on top-level description of SMURF program
	Outlined all acceptable inputs and outputs to a SMURF program
	Assigned vertical slices to team members
10-23-13	Divided backend into semantic analyzer and translator modules instead of single "compiler" module
11-06-13	Decided to add polymorphism back into language
	Discussed structure of Semantic Analyzer modules
11-08-13	Semantic analyzer started
11-13-13	Parser completed
11-20-13	Interpreter Started, changed output of semantic analyzer from sast to beefed up symbol table
11-27-13	Hello World end-to-end compilation succeeds
12-04-13	Semantic Analyzer with tests completed
12-20-13	Interpreters with all tests passing completed
	Interesting SMURF program end-to-end compilation succeeds

5 Architectural Design

5.1 Overview

The SMURF compiler transforms a SMURF program into a playable MIDI file. The compiler first scans the source file, parses the resulting tokens, creates an abstract syntax tree, semantically checks the abstract syntax tree, translates the tree into intermediate code and finally translates this intermediate representation into a MIDI file which can be played in most media players.



5.2 Scanner

The SMURF program file is first passed to the scanner. The scanner matches the string of input characters to tokens and white spaces. The tokens include keywords, constants, and operators used in a SMURF program. All white space characters except new lines are ignored. Any illegal characters are caught and an exception is raised. The tokens are defined with regular expressions and nested comments use a state machine and counter to be resolved. The scanner was built with the ocaml lexer.

5.3 Parser

The tokens generated in the scanner are then used in the parser. The parser matches the string of tokens to a grammar defining the SMURF language. An abstract syntax tree is built as the tokens are matched to the grammar and stored as a program which is defined as a list of declarations. Syntax errors in the SMURF code will be identified during parsing resulting in an exception being raised. The parser is generated using the grammar and the ocaml yacc program.

5.4 Abstract Syntax Tree

The abstract syntax tree is the intermediate representation of a SMURF program after it has been parsed but before it has been semantically checked. The program can easily be transversed by organizing the code into an abstract syntax tree because of its hierarchical nature.

5.5 Semantic Analyzer

The semantic analyzer uses the abstract syntax tree to build a semantic abstract syntax tree which holds more information like scope and type information. Semantic errors are caught during the translation and transversing of the semantic abstract syntax tree. The semantic analyzer walked through the semantic abstract syntax tree twice, first to create the symbol table and then to do checks using the filled symbol table. The second pass of the semantic abstract syntax tree was required because SMURF does not require variables or functions to be defined before they are used.

5.6 Translator

The symbol table is then passed to our translator which evaluates all expressions and creates an intermediate representation that is then converted into MIDI code. Since symbol table contains the expression of all variables and functions all expressions can be evaluated starting from the main declaration without the semantic abstract symbol tree. Errors found during evaluation of expressions are found causing compilations errors. If there are no errors found then an intermediate representation is produced.

5.7 MIDI Converter

The intermediate representation produced from the translator then is translated into MIDI code using the MIDI converter. The MIDI code can then be played.

6 Test Plan

During the development process of SMURF, to let everyone involve in the development as much as possible, we adopt the slice model, i.e., in each developing stage, everyone has a slice of assignment to work on. One problem with this model is that different people needs to work on a same job, so one people's change to the program can easily crash other people's work. As a result, extensive tests to ensure the quality of the software is crucial.

6.1 Testing Levels

6.1.1 Unit Testing

For lexer and parser, we generated a separate executable to test their functionality (parser_test.ml). This executable reads in SMURF programs, analyzes the input files with lexer and parser, generates the abstract syntax trees, and then spits out the information stored in the ast trees.

6.1.2 Integration Testing

We tested the correctness of semantic checking and code generation models together with the lexer and parser models. We generated the toplevel executable with semantic_test.ml for semantic checking, and with toplevel.ml for code generation. The output of semantic checking is the semantic abstract syntax tree, which is the abstract syntax tree with types of every variables resolved. The output of code generation is the bytecode for midi music generation.

We also tested the integration between our bytecode and the midi music generator in java.

6.1.3 System Testing

The end-to-end SMURF compiler accepts SMURF program and generates MIDIs, with the bytecode for MIDI generation as byproduct. For the ASCII bytecode we compare it with golden results to make sure its correctness. We also listen to the MIDIs generated by SMURF with music players to make the sounds are correct.

6.2 Test Suites

The hierarchy for SMURF test cases is shown in (figure 3). In each developing stage, everyone is in charge of a directory holding test cases constructed for the functionality he/she is working on. Every person needs to give the expecting output for his/her test cases in the **exp** directory. A case passes the test if its output is identical with the corresponding output in the **exp** directory. We have a script for testing all the test cases in the toplevel of the directory running all the test cases and comparing the results with the expect results given by every owner of the cases. The script gives the result about how many test cases passed and which test cases failed, if any. Before committing his/her result to the repository, everyone need to make sure the new change passed all the other people's cases. For the occasions that one people's work need to change the output of other people's cases, he/she need to check the changes are as expected, and then generate new expected results for the cases before committing changes to repository.

There are two types of test cases, to pass and to fail. We don't treat them differently during the test. As long as the program is not broken, on one hand, the pass cases should give the same output, On the other hand, the fail cases should give the same exception message as that stored in **exp** directory. We use the convention to name the fail cases starting with *test-fail-* and name the pass cases starting with *test-*.

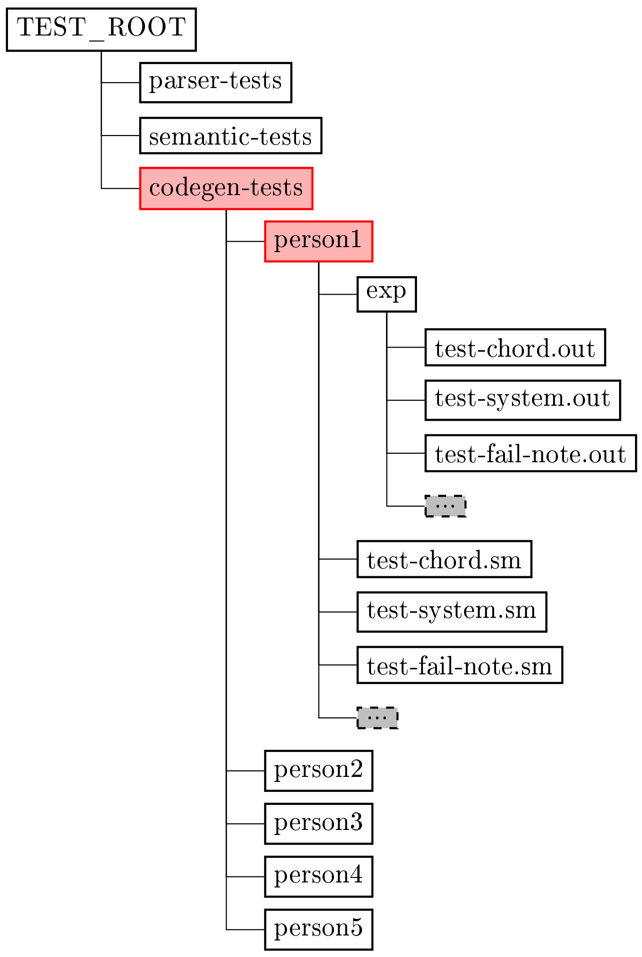


Figure 3: The directory of SMURF test cases

6.3 Example Test Cases

Below we give several sample test cases and their expected output for SMURF.

6.3.1 parser-tests

Note that the programs that pass parser testing may not be semantically correct.

```
a = if 1 then 1 else 2
```

test-if.sm

6.3.2 semantic-tests

For cases successfully passed semantic checking, semantic checking spits out the semantic abstract syntax tree with the type of each variable resolved.

```
a = []

b = [[]]

c = [] ++ []

f = [[[]]] ++ []

d = [1] ++ [2]
```

```
g = [] ++ [1,2,3,4]

h = [True,False,False] : []

i = 1 : []

j = [1,2] : [[1],[2,3]]

aa = [(1,2)$2..] ++ []

bb = [(1,2)$2..] : []

main = []
```

test-emptyList.sm

```
PASSED SEMANTIC CHECKS
```

test-emptyList.out

For cases that failed to passed the semantic checking, semantic checking captures the exception and spits out the message regarding the exception to help the programmer to easily locate the problem in program.

```
a = [(1,2)$4.., (2,3)$2., (3,4)$8., True]

main = []
```

test-fail-chord.sm

```
Fatal error: exception Sast.Type_error("Elements in Chord should all have type of Note but the element of
      true has type of Bool")
```

test-fail-chord.out

6.3.3 codegen-tests

Following is an example used for codegen test.

```
/* Sample SMURF program that should play a cascade :-) */

//[Register] -> [Pitch classses] -> [Durations] -> [Chords]
makeChords :: [Int] -> [Int] -> [Beat] -> [Chord]
makeChords [] _ _ = []
makeChords _ [] _ = []
makeChords _ _ [] = []
makeChords r:restr p:restp d:restd = [(p,r)$d] : (makeChords restr restp restd)

pitches1 = [0,2,4,5,7,9,11,0,-1,0,-1,11,-1,11]
pitches2 = [-1,11,9,-1,8,-1,8,-1,7]
pitches3 = [-1,-1,7,5,-1,5,-1,5,-1,4]
pitches4 = [-1,-1,4,2,-1,2,-1,2,-1,0]
endBeats = [4,4,4,4,4,2]
beats1 = [8,8,8,8,8,8,8,(1 $+ 8)] ++ endBeats
beats2 = [1,8,(2..)] ++ endBeats
beats3 = [1,4,8,(2 $+ 8)] ++ endBeats
beats4 = [1,2,8,4.] ++ endBeats
endReg = [0,2,2,0,2,0,2,0,2]
reg1 = [2,2,2,2,2,2,2,3,0,3,0,2,0,2]
reg2 = endReg
reg3 = 0 : endReg
reg4 = reg3

track1 = makeChords reg1 pitches1 beats1
track2 = makeChords reg2 pitches2 beats2
track3 = makeChords reg3 pitches3 beats3
track4 = makeChords reg4 pitches4 beats4

main = [track1,track2,track3,track4]
```

cascade.sm

And following is the output bytecode generated by SMURF program.

```
***** Generated by SMURF *****
number of trace: 4
Time Resolution (pulses per quarter note),4,
track 1,48,track 2,48,track 3,48,track 4,48,
Tick, Note (0-127), Velocity (0-127), Tick, Note (0-127), Velocity (0-127), Tick, Note (0-127), Velocity (0-127), Tick, Note (0-127), Velocity (0-127),
0,60,90,,,,,,,,,,
2,60,0,,,,,,,,,,
2,62,90,,,,,,,,,,
4,62,0,,,,,,,,,,
4,64,90,,,,,,,,,,
6,64,0,,,,,,,,,,
6,65,90,,,,,,,,,,
8,65,0,,,,,,,,,,
8,67,90,,,,,,,,,,
10,67,0,,,,,,,,,,
10,69,90,,,,,,,,,,
12,69,0,,,,,,,,,,
12,71,90,,,,,,,,,,
14,71,0,,,,,,,,,,
14,72,90,,,,,,,,,,
32,72,0,,,,,,,,,,
,,,16,71,90,,,,,,,
,,,18,71,0,,,,,,,
,,,18,69,90,,,,,,,
,,,32,69,0,,,,,,,
,,,,,,20,67,90,,,,
,,,,,,22,67,0,,,,
,,,,,,22,65,90,,,,
,,,,,,32,65,0,,,,
,,,,,,,,,24,64,90,
,,,,,,,,,26,64,0,
,,,,,,,,,26,62,90,
,,,,,,,,,32,62,0,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
36,72,90,36,68,90,36,65,90,36,62,90,
40,72,0,40,68,0,40,65,0,40,62,0,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
44,71,90,44,68,90,44,65,90,44,62,90,
48,71,0,48,68,0,48,65,0,48,62,0,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
,,,,,,,,,,,,
52,71,90,52,67,90,52,64,90,52,60,90,
60,71,0,60,67,0,60,64,0,60,60,0,
```

cascade.sm

7 Lessons Learned

7.1 Lindsay Neubauer

We had a meeting at the same time every week that lasted between one and two hours that everyone attended. This time set aside to make real progress on project was crucial to our success. In the beginning of the semester we spent all the time discussing LRM-related topics and during the latter half of the semester it was split between discussion and coding. Often being in the same room, even for a short amount of time, while coding was helpful for figuring out the trickier aspects of our language. This was particularly helpful for me because OCaml was my first experience using a functional programming language and having access to others with more experience helped me pick it up quicker.

Another important choice we made was to designate a group leader at the beginning of this project. Our group leader was organized with tasks to discuss or complete in each meeting and helped drive the conversation in a productive way. In addition to this, we had a note taker and a person in charge of our GitHub and Latex environments. It was helpful to have ï¿½go toï¿½ people for questions and concerns that arose throughout the project.

After turning in our LRM we decided to divide each part of our language into slices. Each group member was in charge of a different aspect of our language and implemented that slice for each step of the compiler building process. This ownership of a part of the language was helpful and touching each step of the compiler was very helpful for learning. It also allowed each group member to work throughout the semester regardless of the progress made by others.

The most important learning I had from this project are understanding the functional language paradigm and knowledge on how to implement a compiler from start to finish.

7.2 Kuangya Zhai

First of all, the best lesson I learned from this project is: Finish early. The importance of starting early has been told by numerous previous PLT groups while the importance of finishing early has not. By finishing early I mean you should project the finishing time of the deadline of your project a bit earlier than the actual deadline to allow any exceptions. As is always said, deadline is the first productivity. Your efficiency boosts when the deadline approaches. But there exists the possibility that something unexpected happens and you are not going to be able to finished the project in the due day if your plan is to finish everything in the last day. These situation is common when you are working on a group project. Take our group as an example, we projected everything to be finished on the exact morning of the presentation while it turned out that not everything goes well as expected, so we had to presented an incomplete version which was kind of embarrassing. And the problems was solved on the exact afternoon of the presentation but we had no chance the present it again. Had we project our own deadline one day earlier, I believe the result will be much better.

The second thing, enjoy OCaml. Few people has functional programming background before the PLT class. So it's likely that you will have a steeper learning curve when comparing with learning other programming languages. However, when you got used to the functional style, you will find it's at least as expressive and productive as imperative languages you have got used to. The thing I love the most about functional programming is its type checking system. So you will spend tons of time to get your program to compile. But once after that, your program will likely to give the correct result since most of the bugs have been captured at the compiling stage. Also, the side effect free property of functional program guarantees the robustness of your program, which is especially important when you are working on a teaming working project. OCaml is not purely functional. It also keeps several imperative features which might also be helpful and make your life much easier when comparing with Haskell, the pure functional programming language.

7.3 Lianne Lairmore

The first important lesson I learned was that communication between group members is very important. Having a weekly meeting was very helpful for communication and helped us quickly defined what we wanted in out language. Later in the semester we still met but our meetings were more coding instead of talking. It would have been better to spend more time talking about what we were doing and how far along we were. It probably would have been helpful to do some pair programming in some of the tougher part of the project so that when someone got stuck another person knew who to help them.

The second important lesson I learned was that organization is important. It was important to know what each person was suppose to be doing at one time that way you always knew what was expected of you. Although our group might have balanced the loads more evenly this is hard to do not knowing either how much certain parts of the language are going to take or everyones skills. For example a few people in our group had considerably more experience with functional languages than the others and didn't have as steep a learning curve learning OCaml.

The third important lesson I learned was to make a schedule and stick to it. Our group did half of this. We had a schedule but when we fell behind we didn't push hard to catch up. The first half of the semester we staid on schedule but the second half we fell behind and never was able to catch back up making the end a rush to finish everything.

7.4 Richard Townsend

Weekly meetings are a must! While we had them and they were helpful, we mostly used them to discuss overarching language features and code in the same room. In hindsight, it would have been effective to also use these meetings to discuss how we implemented the various features in the compiler. There were many instances where group members had to work with someone else's code and it was not entirely clear what the original programmer's thought process was. By going over these aspects of the compiler, a lot of time would have been saved for future coding.

It would have also been beneficial for us to assign priorities to different features of our langauge, making sure the higher priority features were up and running before the others. In our case, function application was the highest priority (since SMURF is a functional language), but it was the last feature to be fully implemented, leading to some stress and a less-than-ideal demo for our presentation. This problem would also have been mitigated if we assigned some of our vertical slices to pairs of members as opposed to single members. That way, the two members would keep each other on top of the implementation and deadlines associated with that specific language feature.

Finally, take good notes during Stephen's OCaml series of lectures. If something doesn't make sense during the lecture, talk to him or a TA about it ASAP, as you will probably use that aspect of the language in your compiler at some point. It's imperative, especially with a huge project like a compiler, that you can read another team member's code and understand the basic operations and processes going on.

7.5 Van Bui

Getting used to a functional programming mindset is nontrivial if you are more used to the imperative style of programming. It would have been helpful to practice writing OCaml programs through out the semester. The OCaml programming assignments in class helped with this, but I think more assignments or self-practice beyond that would have been very useful. The beginning of the project is mostly planning and writing and the latter half is a lot of OCaml programming. I would often know in mind exactly how to implement some algorithm imperatively, but would struggle trying to come up with how to write it in OCaml. It is a steep learning curve and requires a lot of practice to get used to the functional programming paradigm.

I also learned that if you need help with something, ask for help and ask early. This prevents the project from getting behind schedule. In my case, the members in the group were very helpful once I asked for help, I just wish I had asked for it earlier. My programming slice, function application, turned out to be much harder than I originally anticipated. Since this was the first end-to-end compiler I helped to write, I had no sense really for the difficulty of implementing different langauge features.

Time management is also a major factor. The project requires a lot of time. So when chosing your courses, chose wisely, and take into account the time required for this project and also your other classes. I was taking Operating Systems, which has a lot of programming projects written in C. So there was quite a bit of context switching between C and OCaml throughout the semester. If you are new to functional programming, writing a lot of C code the same semester might not be such a great idea.

This has been mentioned previously, but I will mention it again to underscore its importance. Pair programming could have been helpful for several reasons: to help with understanding OCaml code written by others, debugging, and to switch off to make sure some slice does not get behind schedule.

8 Appendix

```
{ open Parser
  open Lexing
    let cc = [|0|]
}                           (* Get the Token types *)
(* Optional Definitions *)
(* Rules *)

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let identifier = letter (letter | digit | '_')*

let whitespace = [ ' ' '\t' '\r']

rule token = parse
whitespace                    { token lexbuf } (* White space *)
| "//"                        { nlcomment lexbuf }
| "/*"                        { cc.(0)<-cc.(0)+1; nc1 lexbuf }
| '\\'                        { continue lexbuf }
| '\n'                        { Lexing.new_line lexbuf; NL }
| '&'                         { NL }
| '['                         { LLIST }
| ']'                         { RLIST }
| '+'                         { PLUS }
| '-'                         { MINUS }
| '*'                         { TIMES }
| '/'                         { DIV }
| '%'                         { MOD }
| '<'                         { LT }
| '>'                         { GT }
| "<="                        { LE }
| ">="                        { GE }
| "$+"                        { BPLUS }
| "$-"                        { BMINUS }
| "$*"                        { BTIMES }
| "$/"                        { BDIV }
| "$<"                        { BLT }
| "$>"                        { BGT }
| "$<="                       { BLE }
| "$>="                       { BGE }
| "%+"                        { PCPLUS }
| "%-"                        { PCMINUS }
| "=="                        { EQ }
| '!'                         { NOT }
| "&&"                        { AND }
| "||"                        { OR }
| "++"                        { CONCAT }
| ':'                         { CONS }
| "::"                        { TYPE }
| "->"                        { FUNC }
| '='                         { BIND }
| "^^"                        { TRANS }
| '~'                         { INV }
| "<>"                        { RET }
| '('                         { LPAREN }
| ')'                         { RPAREN }
| ','                         { COMMA }
| '.'                         { PERIOD }
| '$'                         { DOLLAR }
| '_'                         { WILD }
| "let"                       { LET }
| "in"                        { IN }
| "if"                        { IF }
| "then"                      { THEN }
| "else"                      { ELSE }
| "True"                      { BOOLEAN(true) }
| "False"                     { BOOLEAN(false) }
| "Int"                       { INT }
| "Bool"                      { BOOL }
| "Beat"                      { BEAT }
| "Note"                      { NOTE }
| "Chord"                     { CHORD }
| "System"                    { SYSTEM }
| "main"                      { MAIN }
| "print"                     { PRINT }
| identifier as id            { VARIABLE(id) }
| (digit)+ as num             { LITERAL(int_of_string num) }
| eof                         { EOF }
| _ as char { raise (Failure("Illegal character: " ^ Char.escaped char)) }


and nlcomment = parse
'\n'             { lexbuf.lex_curr_pos <- lexbuf.lex_curr_pos - 1; token lexbuf }
| _              { nlcomment lexbuf }

```

```
and continue = parse
'\n'             { Lexing.new_line lexbuf; token lexbuf }
| whitespace     { continue lexbuf }

and nc1 = parse
'/'              { nc2 lexbuf }
| '*'            { nc3 lexbuf }
| '\n'           { Lexing.new_line lexbuf; nc1 lexbuf }
| _              { nc1 lexbuf }

and nc2 = parse
'*'              {cc.(0)<-cc.(0)+1; nc1 lexbuf}
| _              {nc1 lexbuf}

and nc3 = parse
'/'              { if(cc.(0) = 1)
                     then (cc.(0) <- cc.(0)-1;token lexbuf)
                     else (cc.(0)<-cc.(0)-1; nc1 lexbuf)
                 }
| '*'            { nc3 lexbuf }
| _              { nc1 lexbuf }
```

../../Code/scanner.mll

```
%{ open Ast
   open Util
%}

%token NL LET IN IF THEN ELSE INT BOOL EOF
%token BEAT NOTE CHORD SYSTEM MAIN RANDOM PRINT
%token PERIOD DOLLAR
%token LPAREN RPAREN LLIST RLIST COMMA
%token TYPE FUNC
%token PLUS MINUS TIMES DIV MOD BTIMES BDIV BPLUS BMINUS PCPLUS PCMINUS
%token EQ NOT AND OR LT GT LE GE BLT BGT BLE BGE
%token CONCAT CONS BIND
%token INV RET TRANS
%token WILD
%token <int> LITERAL
%token <bool> BOOLEAN
%token <string> VARIABLE

%nonassoc IF THEN ELSE INT BOOL NOTE BEAT CHORD SYSTEM MAIN RANDOM PRINT LET IN
%nonassoc LLIST RLIST COMMA
%nonassoc TYPE FUNC
%right BIND
%nonassoc INV RET
%left TRANS
%left OR
%left AND
%nonassoc NOT
%left EQ
%left LT LE GT GE BLT BGT BLE BGE
%left PLUS MINUS BPLUS BMINUS PCPLUS PCMINUS
%left TIMES DIV BTIMES BDIV MOD
%right CONCAT
%right CONS
%nonassoc DOLLAR
%right PERIOD
%nonassoc LPAREN RPAREN

%start program
%type <Ast.program> program

%%

/* List of declarations, possibly surrounded by NL */
program:
    /* nothing */                   { [] }
|   newlines                        { [] }
|   decs                            { List.rev $1 }
|   newlines decs                   { List.rev $2 }
|   decs newlines                   { List.rev $1 }
|   newlines decs newlines          { List.rev $2 }

newlines:
    NL                              { }
|   newlines NL                     { }

decs:
    dec                              { [$1] }
|   decs newlines dec                { $3 :: $1 }  /* declarations are separated by >= 1 newline */

dec:
    VARIABLE TYPE types              { Tysig($1, [$3]) }  /* variable type-sig only have one type */
|   VARIABLE TYPE func_types         { Tysig($1, List.rev $3) }  /* function type-sig have >= 2 types */
|   VARIABLE BIND expr               { Vardef($1, $3) }
|   VARIABLE patterns BIND expr      { Funcdec{ fname = $1; args = List.rev $2; value = $4 } }
|   MAIN BIND expr                   { Main($3) }

/* types for vars */
types:
    INT                              { TInt }
|   BOOL                             { TBool }
|   NOTE                             { TNote }
|   BEAT                             { TBeat }
|   CHORD                            { TChord }
|   SYSTEM                           { TSystem }
|   LLIST types RLIST                { TList($2) }
|   VARIABLE                         { TPoly($1) }

/* types for functions */
func_types:
    types FUNC types                 { $3 :: [$1] }
|   func_types FUNC types            { $3 :: $1 }

patterns:
    pattern                          { [$1] }
|   patterns pattern                 { $2 :: $1 }

pattern:
    LITERAL                          { Patconst($1) }
|   BOOLEAN                          { Patbool($1) }
|   VARIABLE                         { Patvar($1) }
|   WILD                             { Patwild }
|   LLIST comma_patterns RLIST       { Patcomma(List.rev $2) }
|   pattern CONS pattern             { Patcons($1, $3) }
|   LPAREN pattern RPAREN            { $2 }

comma_patterns:
    /* empty */                      { [] }
|   pattern                          { [$1] }
|   comma_patterns COMMA pattern     { $3 :: $1 }

expr:
    LITERAL                  { Literal($1) }
|   VARIABLE                 { Variable($1) }
|   BOOLEAN                  { Boolean($1) }
|   LPAREN expr RPAREN       { $2 }
|   expr PLUS expr           { Binop($1, Add, $3) }
|   MINUS LITERAL            { Literal(-$2) }
|   expr MINUS expr          { Binop($1, Sub, $3) }
|   expr TIMES expr          { Binop($1, Mul, $3) }
|   expr DIV expr            { Binop($1, Div, $3) }
|   expr MOD expr            { Binop($1, Mod, $3) }
|   expr BDIV expr           { Binop($1, BeatDiv, $3) }
|   expr BTIMES expr         { Binop($1, BeatMul, $3) }
|   expr BPLUS expr          { Binop($1, BeatAdd, $3) }
|   expr BMINUS expr         { Binop($1, BeatSub, $3) }
|   expr PCPLUS expr         { Binop($1, PCAdd, $3) }
|   expr PCMINUS expr        { Binop($1, PCSub, $3) }
|   expr LT expr             { Binop($1, Less, $3) }
|   expr GT expr             { Binop($1, Greater, $3) }
|   expr LE expr             { Binop($1, Leq, $3) }
|   expr GE expr             { Binop($1, Geq, $3) }
|   expr BLT expr            { Binop($1, BeatLess, $3) }
|   expr BGT expr            { Binop($1, BeatGreater, $3) }
|   expr BLE expr            { Binop($1, BeatLeq, $3) }
|   expr BGE expr            { Binop($1, BeatGeq, $3) }

|   expr AND expr            { Binop($1, And, $3) }
|   expr OR expr             { Binop($1, Or, $3) }
|   expr EQ expr             { Binop($1, BoolEq, $3) }

|   expr CONCAT expr         { Binop($1, Concat, $3) }
|   expr CONS expr           { Binop($1, Cons, $3) }
|   expr TRANS expr          { Binop($1, Trans, $3) }

|   NOT expr                 { Prefix(Not, $2) }
|   INV expr                 { Prefix(Inv, $2) }
|   RET expr                 { Prefix(Retro, $2) }
|   expr dots                { Beat($1, $2) }
|   LPAREN
    expr COMMA expr
    RPAREN
    DOLLAR expr              { match $7 with
                                 Literal(_) as e -> Note($2, $4, Beat(e,0))
                               | _ -> Note($2, $4, $7) }
|   IF expr
    THEN expr ELSE expr      { If($2, $4, $6) }
|   LLIST expr_list RLIST    { match $2 with
                                 [] -> List($2)
                               | _ -> (match (List.hd $2) with
                                     Note(_,_,_) -> Chord($2)
                                   | Chord(_) -> System($2)
                                   | _ -> List($2)) }
|   LET program IN expr      { Let($2, $4) }

|   VARIABLE args            { Call($1,$2) }
|   PRINT expr               { Print($2) }

args:
    arg                      { [$1] }
|   args arg                 { $2 :: $1 }

arg:
    LITERAL                  { Arglit($1) }
|   BOOLEAN                  { Argbool($1) }
|   VARIABLE                 { Argvar($1) }
|   LLIST expr_list RLIST    { match $2 with
                                 [] -> Arglist($2)
                               | _ -> (match (List.hd $2) with
                                     Note(_,_,_) -> Argchord($2)
                                   | Chord(_) -> Argsystem($2)
                                   | _ -> Arglist($2)) }
|   LPAREN expr RPAREN       { Argparens($2) }

dots:
    PERIOD         { 1 }
|   PERIOD dots    { $2+1 }

expr_list:
    /* Nothing */  { [] }
|   expr_list_back { List.rev $1 }

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

/*
stmt:
    expr                            { Expr($1) }
|   IF expr THEN stmt ELSE stmt   { If($2, $4, $6) }
*/
```

../../Code/parser.mly

```
type operator = Add | Sub | Mul | Div | Mod |
                Less | Leq | Greater | Geq |
                BeatAdd | BeatSub | BeatDiv | BeatMul |
                BeatLess | BeatLeq | BeatGreater | BeatGeq |
                PCAdd | PCSub |
                BoolEq | And | Or | Concat | Cons | Trans

type prefix_operator = Not | Inv | Retro

(* Not sure if these should be here...doing it for type signature definition *)
type types = TInt | TBool | TNote | TBeat | TChord | TSystem | TList of types |
              TPoly of string

type expr =                                 (* Expressions *)
      Literal of int                        (* 42 *)
    | Boolean of bool                       (* True *)
    | Variable of string                    (* bar *)
    | Beat of expr * int                    (* 2. *)
    | Note of  expr * expr * expr           (* (11, 2)^4. *)
    | Binop of expr * operator * expr       (* a + 2 *)
    | Prefix of prefix_operator * expr      (* ! a == 4 *)
    | If of expr * expr * expr              (* if b == 4 then True else False *)
    | List of expr list                     (* [1,2,3,4] *)
    | Chord of expr list                    (* [(11,3)$4., (5,2)$4.]*)
    | System of expr list                   (* [ [(11,3)$4.,(5,2)$4.], [(-1,0)$2] ]*)
    | Call of string * fargs list           (* foo a *)
    | Let of dec list * expr                (* let x = 4 in x + 2 *)
    | Print of expr                         (* print 3 *)

and dec =                                   (* Declarations *)
    Tysig of string * types list            (* f :: Int -> [Note] -> Bool *)
    | Funcdec of func_decl                  (* f x y = x + y *)
    | Vardef of string * expr               (* x = (2 + 5) : [1,2,3] *)
    | Main of expr                          (* main (f x) + (g x) *)

and func_decl = {                      (* Function Declaration *)
    fname : string;                         (* Function name *)
    args : pattern list;                    (* Pattern arguments *)
    value : expr;                           (* Expression bound to function *)
}

and pattern =                          (* Patterns *)
    Patconst of int                         (* integer *)
    | Patbool of bool                       (* boolean *)
    | Patvar of string                      (* identifier*)
    | Patwild                               (* wildcard *)
    | Patcomma of pattern list              (* [pattern, pattern, pattern, ... ] or [] *)
    | Patcons of pattern * pattern          (* pattern : pattern *)

and fargs =                                 (* Function Arguments *)
      Arglit of int                         (* 42 *)
    | Argbool of bool                       (* True *)
    | Argvar of string                      (* bar *)
    | Argbeat of expr * int                 (* 2. *)
    | Argnote of  expr * expr * expr        (* (11, 2)^4. *)
    | Argchord of expr list                 (* [(11,3)$4., (5,2)$4.] *)
    | Argsystem of expr list                (* [ [(11,3)$4.,(5,2)$4.], [(-1,0)$2] ] *)
    | Arglist of expr list                  (* [farg, farg, farg, ... ] or [] *)
    | Argparens of expr                     (* parenthesized expressions *)

type program = dec list                     (* A program is a list of declarations *)

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Boolean(b) -> string_of_bool b
  | Variable(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      ( match o with
        Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/" | Mod -> "%"
      | BeatAdd -> "$+" | BeatSub -> "$-" | BeatMul -> "$*" | BeatDiv -> "$/"
      | PCAdd -> "%+" | PCSub -> "%-"
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
      | BeatLess -> "$<" | BeatLeq -> "$<=" | BeatGreater -> "$>" | BeatGeq -> "$>="
      | And -> "&&" | Or -> "||" | BoolEq -> "=="
      | Concat -> "++" | Cons -> ":" | Trans -> "^^" )
      ^ " " ^ string_of_expr e2
  | Prefix(o, e) ->
      ( match o with Not -> "!" | Inv -> "~" | Retro -> "<>" )
      ^ " " ^ string_of_expr e
  | If(e1, e2, e3) -> "if " ^ string_of_expr e1 ^ " then " ^ string_of_expr e2 ^
    " else " ^ string_of_expr e3
  | Beat(i1, i2) -> string_of_expr i1 ^
        let rec repeat n s =
            if n>0 then
                repeat (n-1) ("." ^ s)
            else s in repeat i2 ""
  | Note(pc, reg, bt) -> " (" ^ string_of_expr pc ^ ", " ^ string_of_expr reg ^ ")$" ^ (string_of_expr bt
    )
  | List(el) -> "[" ^ (String.concat ", " (List.map string_of_expr el)) ^ "]"
  | Chord(el) -> "[" ^ (String.concat ", " (List.map string_of_expr el)) ^ "]"
  | System(el) -> "[" ^ (String.concat ", " (List.map string_of_expr el)) ^ "]"
  | Call(fname,args) -> fname ^ " " ^ (String.concat " " (List.map string_of_args args))
  | Let(decl, exp) -> "let " ^ (String.concat " " (List.map string_of_dec decl)) ^
                      " in " ^ string_of_expr exp
  | Print(e) -> "print ("^(string_of_expr e)^")"

and string_of_fdec = function
    {fname;args;value} -> fname ^ " " ^  String.concat " "
      (List.map string_of_patterns args) ^ " = " ^ string_of_expr value ^ "\n"

and string_of_dec  = function
    Tysig(id, types) -> id ^ " :: " ^ String.concat "-> " (List.map string_of_types types) ^
      "\n"
  | Vardef(id, expr) -> id ^ " = " ^ string_of_expr expr ^ "\n"
  | Funcdec(fdec) -> string_of_fdec fdec
  | Main(expr) -> "main " ^ string_of_expr expr ^ "\n"

and string_of_patterns  = function
    Patconst(l) -> string_of_int l
  | Patbool(b) -> string_of_bool b
  | Patwild -> "_"
  | Patvar(s) -> s
  | Patcomma(p) -> "[" ^ (String.concat ", " (List.map string_of_patterns p)) ^ "]"
  | Patcons(p1, p2) -> (string_of_patterns p1) ^ " : " ^ (string_of_patterns p2)

and string_of_types  = function
    TInt -> "Int" | TBool -> "Bool" | TChord -> "Chord"
  | TNote -> "Note" | TBeat -> "Beat" | TSystem -> "System"
  | TList(t) -> "[" ^ string_of_types t ^ "]" | TPoly(v) -> "Poly " ^ v

and string_of_args  = function
    Arglit(l) -> string_of_int l
  | Argbool(b) -> string_of_bool b
  | Argvar(s) -> s
  | Arglist(el) ->  "[" ^ (String.concat " " (List.map string_of_expr el)) ^ "]"
  | Argparens(p) -> "(" ^ (string_of_expr p)  ^ ")"
  | Argbeat(i1,i2) -> string_of_expr i1 ^
        let rec repeat n s =
            if n>0 then
                repeat (n-1) ("." ^ s)
            else s in repeat i2 ""
  | Argnote(pc, reg, bt) -> " (" ^ string_of_expr pc ^ ", " ^ string_of_expr reg ^ ")$" ^ (string_of_expr
      bt)
  | Argchord(el) -> "[" ^ (String.concat ", " (List.map string_of_expr el)) ^ "]"
  | Argsystem(el) -> "[" ^ (String.concat ", " (List.map string_of_expr el)) ^ "]"

let string_of_program decs =
  String.concat "" (List.map string_of_dec decs)
```

../../Code/ast.ml

```
(* File: interpeter.ml
 * interpret a file from AST to SMURFy code *)

open Ast
open Sast
open Util
open Printf
open Values
open Output
open Semanalyze

let ticks_16 = [| 1 |]
let ticks_8 = [| 2; 3 |]
let ticks_4 = [| 4; 6; 7 |]
let ticks_2 = [| 8; 12; 14; 15 |]
let ticks_1 = [| 16; 24; 28; 30; 31 |]

let r_max = 1000000

(* conrvent the symbol table defined in Sast to environment defined in Values
 * and set the parent of the new environment to the one passed to it
 *)
(* Values.environment -> Sast.symbol_table -> Values.environment' *)
let st_to_env par st =
    let newmap = List.fold_left (fun mp {v_expr=ve; name=nm; pats=pl} ->
            NameMap.add nm {nm_expr=ve; nm_value=VUnknown} mp)
        NameMap.empty st.identifiers
    in {parent=par; ids=newmap}


(* update a variable 'name' with the value of 'v',
 * looking for the definition of 'name' from the current
 * scope all the way up to the global scope, if can't find the
 * definition in the global scope, add a new binding of
 * 'name' to 'v' in the current scope
 * function returns an updated environment chain
 *)
(* environment -> string -> value -> environment' *)
let rec update_env env name v =
    match NameMap.mem name env.ids with
          true -> let newE = {parent=env.parent;
              ids=NameMap.add name {nm_value=v;nm_expr=None} env.ids} in (*show_env newE;*) newE
```

```
         | false -> match env.parent with
               None -> let newE = {parent=env.parent;
                   ids=NameMap.add name {nm_value=v;nm_expr=None} env.ids} in (*show_env newE;*) newE
             | Some par -> let newE  = {parent=Some (update_env par name v);
                 ids=env.ids} in (*show_env newE;*) newE


(* searching for the definition of a name, returns its value *)
(* environment -> string -> value,environment' *)
let rec resolve_name env symtab name =
    match NameMap.mem name env.ids with
          true -> let id=(NameMap.find name env.ids) in (*print_string ("In resolve_name we found the
    name " ^ name);*)
             (match id.nm_expr with
                   Some expr -> (*print_string ("We found an expr for " ^name ^ "and it is: " ^(
    string_of_sexpr expr) ^ "\n");*)let (v,env1)=(eval env symtab expr) in
                     let env2 = update_env env1 name v in v,env2
                 | None -> (*print_string ("No expr for " ^ name ^ " but we have a value of: " ^ (
    string_of_value id.nm_value) ^ "\n");*) id.nm_value,env)
         | false -> match env.parent with
               None -> interp_error ("Can't find binding to " ^ name)
             | Some par -> resolve_name par symtab name

(* eval : env -> Sast.expression -> (value, env') *)
(* evaluate the expression, return the result and the updated
 * environment, the environment updated includes the
 * current and all the outer environments that modified
 *)
(* environment -> symbol_table -> Sast.s_expr -> (value, environment') *)
and eval env symtab = function
      Sast.SLiteral(x) -> (VInt(x), env)
    | Sast.SBoolean(x) -> (VBool(x), env)
    | Sast.SVariable(str) ->
        let v,env' = resolve_name env symtab str in v,env'
    | Sast.SBeat(e, n) ->  if n < 0 then interp_error ("Somehow we have a negative number of dots on a
    beat!")
        else let (ve,env1) = eval env symtab e in
        (match ve with
            | VInt(x) -> (match x with
                1 -> if n > 4 then interp_error ("A whole Beat may only have up to 4 dots")
                     else (VBeat(ticks_1.(n)),env1)
              | 2 -> if n > 3 then interp_error ("A half Beat may only have up to 3 dots")
                     else (VBeat(ticks_2.(n)),env1)
              | 4 -> if n > 2 then interp_error ("A quarter Beat may only have up to 2 dots")
                     else (VBeat(ticks_4.(n)),env1)
              | 8 -> if n > 1 then interp_error ("An 8th Beat may only have up to 1 dot")
                     else (VBeat(ticks_8.(n)),env1)
              | 16  -> if n > 0 then interp_error ("A 16th Beat may not have dots")
                      else (VBeat(ticks_16.(n)),env1)
              | _ -> interp_error ("Beat must be a power of 2 between 1 & 16"))

            | _ -> interp_error ("Not expected Beat values"))
    | Sast.SNote(pc, reg, beat) ->(*print_string "HERE WE GO";*)
        (let (vpc,env1) = eval env symtab pc in
         let (vreg,env2) = eval env1 symtab reg in
         let (vbeat,env3) = eval env2 symtab beat in let vbeat =
          (match vbeat with
           VBeat(_) -> vbeat
           | VInt(x) -> if List.mem x [1;2;4;8;16] then VBeat(16/x) else interp_error ("Non-power of two
    being used as a beat")
           | _ -> interp_error ("We have a note with a non-int non-beat Beat value")) in (*print_string
    ("Making a note with number of ticks " ^ (string_of_value vbeat));*) VNote(vpc,vreg,vbeat),env3)
    | Sast.SBinop(e1, op, e2) -> (*Incomplete*)
        (let (v1,env1) = eval env symtab e1 in
         let (v2,env2) = eval env1 symtab e2 in
         let ticks = [| 0; 16; 8; 0; 4; 1; 0; 0; 2; 0; 0; 0; 0; 0; 0; 1|] in
         (match v1, v2 with
            | VInt(x), VList(lst) ->
                (match op with
                    Trans -> if not (List.for_all (fun v -> match v with VInt(x) ->
                                        if x >= 0 || x <= 11 then true else false
                                        | _ -> false) lst)
                             then interp_error ("Non pitch class integer found in inversion list")
                             else VList(List.map (fun v -> match v with VInt(y) ->
                                        VInt((x+y) mod 12)
                                        | _ -> interp_error ("Ran into a transposition error"))
                                        lst), env2
                  | Cons -> (match (List.hd lst) with
                      VInt(_) -> VList(v1 :: lst), env2
                      |_ -> interp_error ("Trying to cons an int onto a list of non-ints"))
                  | _ -> (*print_string ("Problem expression: " ^ (string_of_sexpr (Sast.SBinop(e1,op,e2
))) ^ "\n");*) interp_error ("The only op that can be used between an int
                    and a list is the transposition operator"))
        | VInt(x), VInt(y) ->
            (match op with
                  Add -> VInt(x+y),env2
                | Sub -> VInt(x-y),env2
                | Mul -> VInt(x*y),env2
                | Div ->
                    if y<>0
                    then VInt(x/y),env2
                    else interp_error ("Cannot divide by zero")
                | Mod ->
                    if y<0
                    then VInt(x mod (y*(-1))),env2
                    else VInt(x mod y),env2
                | PCAdd -> VInt((x+y) mod 12),env2
                | PCSub -> VInt((x-y) mod 12),env2
                | Less -> VBool(x<y),env2
                | Leq -> VBool(x<=y),env2
                | Greater -> VBool(x>y),env2
                | Geq -> VBool(x>=y),env2
                | BoolEq -> VBool(x=y),env2
                | BeatAdd ->
                    if List.mem x [1;2;4;8;16] && List.mem y [1;2;4;8;16]
                    then (* This is a hacky way of doing this *)
                        VBeat(ticks.(x) + ticks.(y)),env2
                    else interp_error ("Ints used in Beat operation must be power of 2 "
                        ^ "between 1 & 16")
                | BeatSub ->
                    if List.mem x [1;2;4;8;16] && List.mem y [1;2;4;8;16]
                    then (* This is a hacky way of doing this *)
                        if ticks.(x) > ticks.(y)
                        then VBeat(ticks.(x) - ticks.(y)),env2
                        else interp_error ("First operand must be greater than second in Beat
subtraction")
                    else interp_error ("Ints used in Beat operation must be power of 2 "
                        ^ "between 1 & 16")
                | BeatMul ->
                    if y>0 then
                        if List.mem x [1;2;4;8;16]
                        then (* This is a hacky way of doing this *)
                            VBeat(ticks.(x) * y),env2
                        else interp_error ("Ints used in Beat operation must be power of 2 "
                            ^ "between 1 & 16")
                    else interp_error ("Must multiple Beat by positive Int")
                | BeatDiv ->
                    if y>0 then
                        if List.mem x [1;2;4;8;16]
                        then (* This is a hacky way of doing this *)
                            if ticks.(x) > y
                            then VBeat(ticks.(x) / y),env2
                            else interp_error ("First operand must be greater than second in Beat
division")
                        else interp_error ("Ints used in Beat operation must be power of 2 "
                            ^ "between 1 & 16")
                    else interp_error ("Must divide Beat by positive Int")
                | BeatLess ->
                    if List.mem x [1;2;4;8;16] && List.mem y [1;2;4;8;16]
                    then (* This is a hacky way of doing this *)
                        VBool(ticks.(x) < ticks.(y)),env2
                    else interp_error ("Ints used in Beat operation must be power of 2 "
                        ^ "between 1 & 16")
                | BeatLeq ->
                    if List.mem x [1;2;4;8;16] && List.mem y [1;2;4;8;16]
                    then (* This is a hacky way of doing this *)
                        VBool(ticks.(x) <= ticks.(y)),env2
                    else interp_error ("Ints used in Beat operation must be power of 2 "
                        ^ "between 1 & 16")
                | BeatGreater ->
                    if List.mem x [1;2;4;8;16] && List.mem y [1;2;4;8;16]
                    then (* This is a hacky way of doing this *)
                        VBool(ticks.(x) > ticks.(y)),env2
                    else interp_error ("Ints used in Beat operation must be power of 2 "
                        ^ "between 1 & 16")
                | BeatGeq ->
                    if List.mem x [1;2;4;8;16] && List.mem y [1;2;4;8;16]
                    then (* This is a hacky way of doing this *)
                        VBool(ticks.(x) >= ticks.(y)),env2
                    else interp_error ("Ints used in Beat operation must be power of 2 "
                        ^ "between 1 & 16")
                | _ -> interp_error ("Not expected op for Ints"))
        | VBeat(x), VBeat(y) ->
            (* Operations act the same as normal because Beat has been converted to Ticks*)
            (match op with
                  BeatAdd -> VBeat(x+y),env2
                | BeatSub ->
                    if x > y
                    then VBeat(x-y),env2
                    else interp_error ("First operand must be greater than second in Beat subtraction
")
                | BeatLess -> VBool(x<y),env2
                | BeatLeq -> VBool(x<=y),env2
                | BeatGreater -> VBool(x>y),env2
                | BeatGeq -> VBool(x>=y),env2
                | _ -> interp_error ("Not expected op for Beats"))
        | VBeat(x), VInt(y) ->
            (match op with
                | BeatAdd ->
                    if List.mem y [1;2;4;8;16]
                    then VBeat(x + ticks.(y)),env2
                    else interp_error ("Ints used in Beat operation must be power of 2 " ^
                        "between 1 & 16")
                | BeatSub ->
                    if List.mem y [1;2;4;8;16] then
                        if x > ticks.(y)
                        then VBeat(x - ticks.(y)),env2
                        else interp_error ("First operand must be greater than second in Beat
subtraction")
                    else interp_error ("Ints used in Beat operation must be power of 2 " ^
                        "between 1 & 16")
                | BeatMul ->
                    if y>0
                    then VBeat(x*y),env2
                    else interp_error ("Must multiple Beat by positive Int")
                | BeatDiv ->
                    if y>0 then
                        if x>y
                        then VBeat(x/y),env2
                        else interp_error ("First operand must be greater than second in Beat
division")
                    else interp_error ("Must divide Beat by positive Int")
                | BeatLess ->VBool(x < ticks.(y)),env2
                | BeatLeq ->VBool(x <= ticks.(y)),env2
                | BeatGreater -> VBool(x > ticks.(y)),env2
                | BeatGeq -> VBool(x >= ticks.(y)),env2
                | _ -> interp_error ("Not expected op for Beats"))
        | VInt(x), VBeat(y) -> if not (List.mem x [1;2;4;8;16]) then
                               interp_error ("Ints used in Beat operation must be power of 2 " ^ "
between 1 & 16")
                               else
            (match op with
                | BeatAdd -> VBeat(ticks.(x) + y),env2
                | BeatSub -> if ticks.(x) > y
                            then VBeat(ticks.(x) - y),env2
                            else interp_error ("First operand must be greater than second in Beat
subtraction")
                | BeatLess -> VBool(ticks.(x) < y),env2
                | BeatLeq -> VBool(ticks.(x) <= y),env2
                | BeatGreater -> VBool(ticks.(x) > y),env2
                | BeatGeq -> VBool(ticks.(x) >= y),env2
                | _ -> interp_error ("Not expected op for Beats"))
        | VBool(x), VBool(y) ->
            (match op with
                  And -> VBool(x && y),env2
                | Or -> VBool(x || y),env2
                | _ -> interp_error ("Not expected op for Bools"))
        | VList([]), x ->
            (match x with
                 VList(m) ->
                    (match op with
                        Concat -> VList(m),env2
                       | Cons -> (match m with
                                  [VSystem(sys)] ->
                                    VList([VSystem(VChord([VNote(VInt(-1),VInt(-1),VBeat(-1))])::sys)
]),env2
                                  | _ -> VList(VList([])::m),env2)
                       | _ -> interp_error ("Not expected op between empty list and List"))
                 | VChord(m) -> (match op with
                        Concat -> VChord(m),env2
                        | Cons -> VChord((VNote(VInt(-1),VInt(-1),VBeat(-1))) :: m),env2
                        | _ -> interp_error ("Not expected op between empty list and Chord"))
                 | VSystem(n) -> (match op with
                         Concat -> (VSystem(n),env2)
                       | Cons -> (VSystem(VChord([VNote(VInt(-1),VInt(-1),VBeat(-1))])::n),env2)
                       | _ -> interp_error ("Not expected op between empty list and System"))
                 |_ -> interp_error("Empty list being applied to nonlist operand in binary operation"
))
        | x, VList([]) ->
            (match op with
                Concat -> x, env2
              | Cons -> VList([x]), env2
              | _ -> interp_error ("Not expected op given two lists with second being the empty list"
))
        | VList(lx), VList(ly) ->
            (match op with
                  Concat -> VList(lx @ ly),env2
                | Cons -> (match (List.hd ly) with
                            VList(_) -> VList(v1 :: ly),env2
                           |VChord(_) -> (match (List.hd lx) with
                                           VNote(_,_,_) -> VList(v1 :: ly)
                                         | _ -> interp_error ("Cannot cons non-note " ^ (
string_of_value v1) ^ " onto chord")),env2
                           | _ -> interp_error ("Cannot cons " ^ (string_of_value v1) ^ " onto " ^ (
string_of_value v2)))
                | _ -> interp_error ("Not expected op for Lists: " ^ (string_of_value v1) ^ " " ^ (
string_of_value v2)))
        | VNote(a,b,c), (VList(lst) | VChord(lst)) -> (match op with
                Cons -> let notetester = (fun note-> match note with VNote(d,e,f) -> f =c | _ ->
false) in
                        (match (List.hd lst) with
                            VNote(_,_,_) -> (match v2 with
                                                VChord(_) -> if List.for_all notetester lst then
VChord(v1 :: lst)
                                                            else interp_error ("One of the notes in "
 ^ (string_of_value v2) ^
                                                                " does not have the same duration as
" ^ (string_of_value v1))

                                               | _ -> VList(v1 :: lst)), env2
                          | _ -> interp_error ("Cannot cons a note to a list of non-notes"))
              | _ -> interp_error ("Not expected op given a note and a list"))
        | VChord(a), VList(lst)-> (match op with
                Cons -> (match (List.hd lst) with
                    VChord(_) -> VList(v1 :: lst), env2
                  | VList(VNote(_,_,_) :: _) -> VList(v1 :: lst), env2
                  | _ -> interp_error ("Cannot cons a chord to a list of non-chords"))
               | _ -> interp_error ("Note expected op given a chord and a list"))
        | VChord(a), VSystem(lst) -> (match op with
                Cons -> VSystem(v1 :: lst), env2
               | _ -> interp_error ("Note expected op given a chord and a system"))
        | x, y ->
            (match op with
                  BoolEq -> VBool(x=y),env2
                | _ -> interp_error ((string_of_value x) ^ " " ^(string_of_value y) ^ ": Not expected
 operands")))
     )
| Sast.SPrefix(op, e) -> (*Incomplete*)
    (let (v1,env1) = eval env symtab e in
     match v1 with
        | VBool(x) -> (match op with
            | Not -> VBool(not x),env1
            | _ -> interp_error ("Unexpected op for Bool"))
        | VList(lst) -> (match op with
            | Retro -> VList(List.rev lst),env1
            | Inv -> if List.for_all (fun v -> match v with VInt(x) ->
                                    if x >= 0 || x <= 11 then true
                                    else interp_error ("Non pitch class integer found in inversion list")
                                   | _ -> false) lst then
                     let row = List.map (fun v -> match v with
                            VInt(x) -> x
                          | _ -> interp_error("Non int found in a list of int")) lst in
                     let base = List.hd row in
                     let transrow = List.map (fun v -> v - base) row in
                     let invrow = List.map (fun v -> 12 - v) transrow in
                     let finalrow = List.map (fun v -> v + base) invrow in
                     VList(List.map (fun v -> VInt(v)) finalrow), env1
                     else interp_error ("Inversion called on non-tone row")

            | _ -> interp_error ("Unexpected op for list"))
        | _ -> interp_error ("Unexpected operand for prefix op")
     )
| Sast.SIf(e1, e2, e3) ->
    (match eval env symtab e1 with
            | VBool(true), env -> eval env symtab e2
            | VBool(false), env -> eval env symtab e3
            | _ -> interp_error ("error in If expr"))
    | Sast.SList(el) -> if el = [] then  (VList([]), env) else  (*updating evironment after eval every
    expression*)
       (let (env',lst)=(List.fold_left (fun (env,lst) e ->
                   let v, env' = eval env symtab e in (env',v::lst))
               (env,[]) el) in VList(List.rev lst), env')
    | Sast.SChord(el) ->
       (let (env',lst)=(List.fold_left (fun (env,lst) e ->
                   let v, env' = eval env symtab e in (env',v::lst))
               (env,[]) el) in VChord(List.rev lst), env')
    | Sast.SSystem(el) ->
       (let (env',lst)=(List.fold_left (fun (env,lst) e ->
                   let v, env' = eval env symtab e in (env',v::lst))
               (env,[]) el) in VSystem(List.rev lst), env')

    | Sast.SCall(e1, e2) ->
       let sid_lst = List.find_all (fun f -> f.name = e1) symtab.identifiers in
       let sid =
       try
       List.find (fun sid -> let flag,_ = bind_pat_arg env symtab sid.pats e2 in flag) sid_lst
       with Type_error x -> type_error x
           | Not_found -> interp_error ("Matched patern not found!")
       in
       let flag,newE = bind_pat_arg env symtab sid.pats e2 in
       (match sid.v_expr with
         Some(e) -> (*print_string ("The expression we ended up with is: "^(string_of_sexpr e)^"\n"); *)
                   (match (eval newE symtab e) with v, _ -> v,env )
       | None -> (*print_string "WE GOT NONE\n"; *)interp_error ("Function declaration without
    expression"))

    | Sast.SLet(s_prog,e) -> (* reutrn the original env *)
       let local_env = st_to_env (Some env) s_prog.symtab in
       let local_env1 = List.fold_left exec_decl local_env s_prog.decls in
       show_env local_env1; let v,local_env2 = (eval local_env1 symtab e) in v,env
    | Sast.SRandom -> Random.self_init (); (VInt(Random.int r_max), env)
    | Sast.SPrint(e1) -> (*print_string ("\n"^(string_of_value (fst (eval env symtab e1)))^"\n") ;*) eval
     env symtab e1


(* environment -> pattern list -> arg list -> (Bool,environment') *)
and bind_pat_arg env symtab patl argl =
    if (List.length patl) <> (List.length argl) then
         type_error ("number of arguments does not match number of patterns")
    else
    (*print_string (String.concat " " (List.map string_of_patterns patl));
    print_string ("\n" ^ String.concat " and " (List.map string_of_sfargs (List.rev argl)) ^ "\n");*)
    let combl = List.combine patl (List.rev argl) in
    let flag,nmp = List.fold_left (fun (flag,mp) (p,a) -> let b,mp' = is_pat_arg_matching env symtab p a
    mp in (*print_string (string_of_bool b); *)
            (flag&&b,mp')) (true,NameMap.empty) combl
    in (*print_string "RETURNING FROM BIND_PAT\n";*) flag,{parent=Some(env); ids=nmp}

and gen = function
    _ as v -> {nm_expr = None; nm_value=v}

(* pattern -> value -> NameMap -> (Bool, NameMap') *)
and is_pat_val_matching env symtab pat value mp =
    match pat with
          Patconst(pi) -> (match value with
                           VInt(ai) -> if pi = ai then true,(mp) else false,mp
                         | _ -> false,(mp))
        | Patbool(pb) -> (match value with
                           VBool(ab) -> if pb = ab then true,(mp) else false,mp
                         | _ -> false,mp)
        | Patvar(ps) -> (*print_string "IN PAT_VAL MATCHING\n\n\n";*)(match value with
                           VInt(ai) -> true,(NameMap.add ps (gen (VInt(ai))) mp)
                         | VBool(ab) -> true,(NameMap.add ps (gen (VBool(ab))) mp)
                         | VBeat(i) -> true, (NameMap.add ps (gen value) mp)
                         | VNote(_,_,_) | VChord(_) | VSystem(_) | VList(_) -> true,(NameMap.add ps (gen
    value) mp)
                         | _ -> interp_error ("We have an unknown value in the interpreter...\n"))
        | Patwild -> true,(mp)
        | Patcomma(pl) -> (match value with
                         | VList(al) | VChord(al) | VSystem(al) -> (if List.length pl <> List.length al
    then
                             false,mp
                             else
                             let lst = List.combine pl al in
                             List.fold_left (fun (b,m) (p,a) -> let r1,r2 = match_pat_value env symtab p a
      m in (b&&r1),r2) (true,mp) lst)
                         | _ ->  false,mp)
        | Patcons(p1,p2) -> (match value with
                         | VList(al) -> (if List.length al = 0 then
                             false,mp
                             else
                             (match al with
                               h::tl ->
                                 (let r1,r2 = match_pat_value env symtab p1 h mp in
                                  let r3,r4 = is_pat_val_matching env symtab p2 (VList(tl)) r2 in (r1&&r3)
     ,r4)
                             | _ -> false,mp))
                         | _ -> false,mp)

(* pattern -> argument -> NameMap -> (Bool,NameMap') *)
and is_pat_arg_matching env symtab pat arg mp =
    match pat with
          Patconst(pi) -> (match arg with
                           SArglit(ai) -> if pi = ai then true,(mp) else false,mp
                         |  SArgparens(expr) -> (let v,_ = eval env symtab expr in
                                                 (match v with
                                                  VInt(ai) -> if pi = ai then true, (mp) else false,mp
                                                 | _ -> false,mp))
                         | SArgvar(id) -> let v,_ = resolve_name env symtab id in if (match v with
                                                     VInt(ai) -> pi = ai
                                                    | _ -> false) then true, (mp) else false,mp
                         | _ -> false,(mp))
        | Patbool(pb) -> (match arg with
                           SArgbool(ab) -> if pb = ab then true,(mp) else false,mp
                         | _ -> false,mp)
        | Patvar(ps) -> (*print_string "Patvar\n"; *)(match arg with
                           SArglit(ai) -> true,(NameMap.add ps (gen (VInt(ai))) mp)
                         | SArgbool(ab) -> true,(NameMap.add ps (gen (VBool(ab))) mp)
                         | SArgvar(str) -> (*print_string ("In is_pat_arg_matching we're trying to match
     pattern " ^ (string_of_patterns (Patvar(ps))) ^
                                                         " with argument " ^ str);*) let v,_ =
     resolve_name env symtab str in true,(NameMap.add ps (gen v) mp)
                         | SArgbeat(e,i) ->
                             (match (eval env symtab (SBeat(e,i))) with
                                (VBeat(aa),_) -> true,(NameMap.add ps (gen (VBeat(aa))) mp)
                              | _ -> false,(mp))
                         | SArgnote(p,r,b) ->
                             (match eval env symtab (SNote(p,r,b)) with
                                VNote(v1,v2,v3),_ -> true,(NameMap.add ps (gen (VNote(v1,v2,v3))) mp)
                              | _ -> false,mp)
                         | SArgchord(el) ->
                             (let vl,env = List.fold_left (fun (l,env) e -> let res,env' = eval env symtab
      e in (res::l),env') ([],env) el in
                              true,(NameMap.add ps (gen (VChord(vl))) mp))
                         | SArgsystem(el) ->
                             (let vl,env = List.fold_left (fun (l,env) e -> let res,env' = eval env symtab
      e in (res::l),env') ([],env) el in
                              true,(NameMap.add ps (gen (VSystem(vl))) mp))
                         | SArglist(el) ->
                             (let vl,env = List.fold_left (fun (l,env) e -> let res,env' = eval env symtab
      e in (res::l),env') ([],env) el in
                              true,(NameMap.add ps (gen (VList(vl))) mp))
                         | SArgparens(expr) -> (*print_string ("Dealing with parens expr: " ^ (
     string_of_sexpr expr) ^ "\n");*)
                             (let v,_ = eval env symtab expr in true,(NameMap.add ps (gen v) mp)))
        | Patwild -> true,(mp)
        | Patcomma(pl) -> (*print_string "Patcomma\n";*)(match arg with
                         | SArglist(al) -> (if List.length pl <> List.length al then
                             false,mp
                             else
                             let lst = List.combine pl al in
                             List.fold_left (fun (b,m) (p,a) -> let r1,r2 = match_pat_expr env symtab p a
     m in (b&&r1),r2) (true,mp) lst)
                         | SArgvar(id) -> (*print_string ("For patcomma, we have an argument var named " ^
      id ^ "\n");*)
                                          let v,_ = resolve_name env symtab id in (match v with
                                              VList(lst) -> (match lst with
                                                            | [] -> if pl = [] then true,(mp) else false,(
     mp)
                                                            | hd::tl -> if pl = [] then false,(mp) else
                                                                 (let r1,r2 = match_pat_value env symtab (
     List.hd pl) hd mp in
                                                                  let r3,r4 = is_pat_val_matching env
     symtab (Patcomma((List.tl pl))) (VList(tl)) r2 in (r1 &&r3),r4))
                                            | _ -> interp_error ("Not working right now"))
                         | _ -> false,mp)
        | Patcons(p1,p2) -> (match arg with
                        | SArglist(al) | SArgchord(al) | SArgsystem(al)-> (if List.length al = 0 then
                            false,mp
                            else
                            (match al with
                              h::tl ->
                                (let r1,r2 = match_pat_expr env symtab p1 h mp in
                                 let r3,r4 = is_pat_arg_matching env symtab p2
                                 (match arg with
                                  SArglist(_) -> SArglist(tl)
                                 |SArgchord(_) -> SArgchord(tl)
                                 |SArgsystem(_) -> SArgsystem(tl)
                                 | _ -> interp_error("Not acceptable")) r2 in (r1&&r3),r4)
                            | _ -> false,mp))
                        | SArgvar(id) -> (*print_string ("For patcons, we have an argument var named " ^
    id  ^ "\n");*)
                                         let v,_ = resolve_name env symtab id in (match v with
                                             VList(lst) | VChord(lst) | VSystem(lst) -> (match lst with
                                                            h::tl ->
                                                                (let r1,r2 = match_pat_value env symtab
    p1 h mp in
                                                                 let r3,r4 = is_pat_val_matching env
    symtab p2
                                                                 (match v with
                                                                  VList(_) -> VList(tl)
                                                                 |VChord(_) -> VChord(tl)
                                                                 |VSystem(_) -> VSystem(tl)
                                                                 | _-> interp_error("Not acceptable"))
                                                                  r2 in (r1 &&r3),r4)
                                                           | _ -> false, mp)
                                             | _ ->  false, mp)
                        | SArgparens(exp) -> let r1,r2 =match_pat_expr env symtab pat exp mp in r1,r2
                        | _ -> false,mp)
    (*
        *)


and match_pat_expr env symtab pat expr mp =
    let arg,env = eval env symtab expr in (*print_string (string_of_value arg);*)
    match pat with
          Patconst(pi) -> (match arg with
                          VInt(ai) -> if pi = ai then true,(mp) else false,mp
                        | _ -> false,(mp))
        | Patbool(pb) -> (match arg with
                          VBool(ab) -> if pb = ab then true,(mp) else false,mp
                        | _ -> false,mp)
        | Patvar(ps) -> true,(NameMap.add ps (gen arg) mp)
        | Patwild -> true,(mp)
        | Patcons(p1,p2) -> (match arg with VList(lst) -> (match lst with
                                                            h::tl ->
                                                                (let r1,r2 = match_pat_value env symtab
    p1 h mp in
                                                                 let r3,r4 = is_pat_val_matching env
    symtab p2 (VList(tl)) r2 in (r1 &&r3),r4)
                                                           | _ -> false, mp)
                                             | _ ->  false, mp)

        | _ -> false,mp


(* same as match_pat_expr but matches pattern against value, which occurs when we're comparing
    a list pattern with a variable argument in is_pat_arg_matching *)
and match_pat_value env symtab pat value mp =
    match pat with
      Patconst(pi) -> (match value with
                      VInt(ai) -> if pi = ai then true,(mp) else false,mp
                    | _ -> false,(mp))
    | Patbool(pb) -> (match value with
                      VBool(ab) -> if pb = ab then true,(mp) else false,mp
                    | _ -> false,mp)
    | Patvar(ps) -> (*print_string "TRUE!";*)true,(NameMap.add ps (gen value) mp)
    | Patwild -> true,(mp)
    | _ -> false,mp



(* exec_decl : env -> decl -> env' *)
(* execute the top-level declaration, in the global enviroment,
 * return the updated global environment. Seems several decls needn't
 * be execed as we only evaluate the dependencies of main *)
(* environment -> Sast.s_dec -> environment' *)
```

```
and exec_decl env = function
    (*
      Sast.STypesig(sid) -> (* signature will generate a new fun *)
        (let vfun = VFun(sid.name,s_id,[]) in update_env env str vfun)
    | Sast.SFuncdec(f_decl) -> (* fun decl will be added to current *)
        (match NameMap.mem f_decl.fname env.ids with
              true -> (match (NameMap.find f_decl.fname env.ids) with
                        | {nm_value=VFun(name,fsig,def)} ->
                            let vfun = VFun(name, fsig, f_decl::def) in update_env env name vfun
                        | _ -> interp_error("Not defined as a signature"))
            | false -> interp_error ("Function definition without a signature"))
    | Sast.SVardef(sid,se) ->
        let v,env' = eval env se in
            update_env env' sid.name v
    | Sast.SMain(e) ->
        (let v, env' = eval env e in
            write_to_file bytecode_name v; update_env env' "main" v)
    *)
    | _ -> trace ("Unsupported!") env


(* The entry of evaluation of the program *)
(* environment -> configuration -> unit *)
let exec_main symtab config =
    let globalE=(st_to_env None symtab) in
    let main_entry =NameMap.find "main" globalE.ids in
    let main_expr = (match main_entry.nm_expr with
              None -> interp_error "main has no definition!"
            | Some expr -> expr) in
    let v, env' = eval globalE symtab main_expr in
    let _ = write_to_file config.bytecode_name v; update_env env' "main" v in
    let cmd = ("java -jar " ^ config.lib_path ^ " " ^ config.bytecode_name ^ " " ^ config.midi_name) in
    print_string (cmd ^ "\n");
    let result_code = Sys.command cmd
    in (match result_code with
           0 ->
             print_string ("===== Program Successfully Finished =====\n");
             print_string ("===== Result Writen to " ^ config.midi_name ^ " =====\n")
        | _ as error_code -> print_string ("Error: *** Program Terminates With Code " ^ string_of_int
     error_code ^ "\n")
        )


(* run : program -> () *)
(* run the program. original one, depreciated *)
let run program s_prog =

let decls = program in
let globalE = {parent = None;
        ids = List.fold_left (fun mp lst ->
        NameMap.add lst.name {nm_value=VUnknown; nm_expr=None} mp)
        NameMap.empty s_prog.symtab.identifiers}
in let _ = show_env globalE in

(* top-level declarations always run in global environment *)
List.fold_left exec_decl globalE decls
```

../../Code/interpreter.ml

```
open Ast
open Printf
open Util
open Values
open Random

exception Output_error of string
let output_error msg = raise (Output_error msg)

let default_velocity = 90

(* Write the head of each smurf file, returns the number of tracks *)
(* write_head : out_channel -> value -> int *)
let write_head oc value =
    let header = "***** Generated by SMURF *****" in
    let number_of_track = (match value with
           VList(lst) -> (
                try
                    match List.hd lst with
                      VSystem(_) |  VList(VChord(_)::_) |
                      VList(VList(VNote(_,_,_)::_)::_)-> List.length lst (* list of system *)
```

```
                   | _ -> 1
               with
                 Failure _ -> 0 (* Empty list *)
               )
         | _ -> 1) in
     let resolution = 4 in
     fprintf oc "%s\n" header;
     fprintf oc "number of trace: %d\n" number_of_track;
     fprintf oc "Time Resolution (pulses per quarter note),%d,\n" resolution;
     Random.init 0;
     for i=1 to number_of_track
         do fprintf oc "track %d,%d," i 48
         done;
     fprintf oc "\n";
     for i=1 to number_of_track
         do fprintf oc "Tick, Note (0-127), Velocity (0-127), "
         done;
     fprintf oc "\n";
     number_of_track

(*
(* get the number of ticks of a beat *)
(* VBeat -> Int *)
let ticks_of_beat = function
      VBeat(VInt(i1), -1) -> i1
     | VBeat(VInt(i1),i2) ->
      (int_of_float
          ((2.0 *. (16.0/.(float_of_int i1))) -. ((16.0/.float_of_int i1) /.
              ((match i2 with
                 0 -> 1.0
               | 1 -> 2.0
               | 2 -> 4.0
               | 3 -> 8.0
               | 4 -> 16.0
               | _ -> output_error ("Error in ticks_of_beat: Not valid numbers"))
                        ))))
    | _ -> output_error ("Error in ticks_of_beat: Not a beat")
*)

(* figure how many ticks are there in the output, so that an array with suitable size can be generated *)
(* value -> Int *)
let rec ticks_of_output value =
    match value with
      VNote(pc,reg,beat) ->
      (match beat with
          | VBeat(-1) -> 0
          | VBeat(1) -> 2
          | VBeat(beat) -> beat
          | VInt(beat) -> beat
          | _ -> interp_error ("Invalid Beat value")
      )
    | VChord(nlst) -> List.fold_left (fun acc ch -> acc + ticks_of_output ch) 0 nlst
    | VSystem(slst) | VList((VList(VList(VNote(_,_,_) :: _) :: _) :: _) as slst)
    | VList((VList(VChord(_) :: _) :: _) as slst)
    | VList((VSystem(_) :: _) as slst) | VList((VList(VNote(_,_,_) :: _) :: _)  as slst) -> List.
     fold_left (fun acc ch -> acc + ticks_of_output ch) 0 slst
    | VList((VNote(_,_,_) :: _) as nlst) | VList((VChord(_) ::_) as nlst) -> List.fold_left (fun acc ch
    -> acc + ticks_of_output ch) 0 nlst
    | _ ->  output_error ("Error in ticks_of_output")


(* Write a note into an array, return the next postion to be writen, and the next tick to begin with *)
(* Value -> Array -> Int -> Int -> Int -> (Int, Int, Int) *)
let rec write_to_array value arr ix iy tic =
    (match value with
    | VNote(VInt(pc),VInt(reg),VBeat(beat)) ->if (pc = -1 && reg = -1 && beat = -1) then ix,iy,tic else
        let note = (match pc with
              -1 -> -1
            | _ -> pc+12*(reg+3)) in (
            arr.(ix).(iy) <- tic;                      (* tick *)
            arr.(ix).(iy+1) <- note;                   (* note *)
            arr.(ix).(iy+2) <- default_velocity;       (* velocity *)
            arr.(ix+1).(iy) <- tic+beat;
            arr.(ix+1).(iy+1) <- note;
            arr.(ix+1).(iy+2) <- 0;
            if beat = 1 then ix+beat+1,iy,tic+beat else ix+beat,iy,tic+beat)
    (* All notes in a chord should fills same set of ticks *)
    | VChord((VNote(_,_,VBeat(ticks))::xs) as nlst) | VList((VNote(_,_,VBeat(ticks))::xs) as nlst) ->
        let actlst = if ticks = -1 then List.tl nlst else nlst in
        (let resx, resy, restic =
         (List.fold_left (fun (x,y,ntic) note ->
            let (nx,ny,ntic) = write_to_array note arr x y ntic
            in (nx,ny,tic)) (ix,iy,tic) actlst) in resx, resy, (if ticks = -1 then restic else restic+
ticks))
    | VSystem(clst) | VList((VChord(_) :: _) as clst) | VList((VList(VNote(_,_,_) :: _) :: _) as clst)->
        (let resx, resy, resz =
        List.fold_left (fun (x,y,ntic) chord ->
            let (nx,ny,ntic) = write_to_array chord arr x y ntic
            in (nx,ny,ntic)) (ix,iy,tic) clst in (0,resy+3,0))
    | VList((x::xs) as slst) -> (match x with
          VSystem(_) | VChord(_) | VNote(_,_,_) | VList(VChord(_)::_) | VList(VList(VNote(_,_,_)::_)::_)
    ->
                    List.fold_left (fun (x,y,ntic) sys ->
                  let (nx,ny,ntic) = write_to_array sys arr x y ntic
                  in (nx,ny,ntic)) (ix,iy,tic) slst
        | _ ->  output_error ("Error in write_to_array: Expression bound to MAIN must "
                            ^ "be the empty list, a note, or a list of systems, chords, or notes"))
    | _ -> output_error ("Error in write_to_array: Input is not a valid value")
    )


(* Write a Chord or a System or a list of Systems to file with smurf specified format *)
(* write_to_file : string -> value -> unit *)
let write_to_file filename value =
    let oc = open_out filename in
    let number_of_track = write_head oc value in
    match number_of_track with
          0 -> close_out oc; print_string ("===== main = [] Program Exits Normally =====\n"); exit 0
        | _ -> (
    let dimx = ticks_of_output value in
    let dimy = number_of_track * 3 in
    let resArr = (Array.make_matrix (dimx+1) (dimy) (-1)) in
    let _ = (write_to_array value resArr 0 0 0) in
    for i=0 to dimx-1 do
        for j=0 to (number_of_track -1) do
            if resArr.(i).(3*j+1) <> (-1) then
                ignore(fprintf oc "%d,%d,%d," resArr.(i).(3*j) resArr.(i).(3*j+1) resArr.(i).(3*j+2))
            else
                ignore(fprintf oc ",,,")
        done;
        ignore(fprintf oc "\n")
    done;
    close_out oc
    )
```

../../Code/output.ml

```

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  let listing = Ast.string_of_program program
  in print_string listing
```

../../Code/parser_test.ml

```
open Ast
open Util

exception Multiple_declarations of string
exception Multiple_type_sigs of string
exception Multiple_patterns of string
exception Pattern_list_type_mismatch of string
exception Cons_pattern_type_mismatch of string
exception Multiple_identical_pattern_lists of string
exception No_type_signature_found of string
exception No_func_dec of string
exception Pattern_num_mismatch of int * int
exception Type_mismatch of string
exception Main_wrong_scope
exception Main_type_mismatch of string
exception Main_missing
exception Function_used_as_variable of string
exception Missing_variable_definition of string
exception Function_not_defined of string
exception Wrong_number_of_arguments of string
exception Function_arguments_type_mismatch of string
exception Type_error of string
let type_error msg = raise (Type_error msg)
```

```
type s_type = Int | Bool | Note | Beat | Chord | System | List of s_type |
              Poly of string | Unknown | Num | Still_unknown | Empty
type s_program = {
    mutable decls : s_dec list;
    symtab : symbol_table;
}

and s_expr =
      SLiteral of int                            (* 42 *)
    | SBoolean of bool                           (* True *)
    | SVariable of string                        (* bar *)
    | SBeat of s_expr * int                        (* 2. *)
    | SNote of  s_expr * s_expr * s_expr               (* (11, 2)^4. *)
    | SBinop of s_expr * operator * s_expr           (* a + 2 *)
    | SPrefix of prefix_operator * s_expr        (* ! a == 4 *)
    | SIf of s_expr * s_expr * s_expr                  (* if b == 4 then True else False *)
    | SList of s_expr list                       (* [1,2,3,4] *)
    | SChord of s_expr list                      (* [(11,3)$4., (5,2)$4.]*)
    | SSystem of s_expr list                     (* [ [(11,3)$4.,(5,2)$4.], [(-1,0)$2] ]*)
    | SCall of string * s_arg list               (* foo a b *)
    | SLet of s_program * s_expr                 (* let x = 4 in x + 2 *)
    | SRandom
    | SPrint of s_expr

and s_arg =
    SArglit of int                               (* integer *)
    | SArgbool of bool                           (* boolean *)
    | SArgvar of string                          (* identifiers *)
    | SArgbeat of s_expr * int                   (* 2. *)
    | SArgnote of  s_expr * s_expr * s_expr      (* (11, 2)^4. *)
    | SArgchord of s_expr list                   (* [(11,3)$4., (5,2)$4.] *)
    | SArgsystem of s_expr list                  (* [ [(11,3)$4.,(5,2)$4.], [(-1,0)$2] ] *)
    | SArglist of s_expr list                    (* expression *)
    | SArgparens of s_expr                       (* parenthesized expressions *)

and s_dec =
     SFuncdec of s_func_decl
    | SVardef of s_ids * s_expr
    | SMain of s_expr

and  s_func_decl = {
    s_fname : string;
    type_sig : s_type list;
    s_args :  pattern list;
    s_value : s_expr;
    scope : symbol_table;
}


and s_ids = {
        name : string;
        pats : pattern list;
        v_type : s_type list;
        v_expr : s_expr option;
}

and symbol_table = {
    parent : symbol_table option;
    mutable identifiers :  s_ids list;
}




let rec string_of_sexpr = function
    SLiteral(l) -> string_of_int l
  | SBoolean(b) -> string_of_bool b
  | SVariable(s) -> s
  | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^
      ( match o with
        Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/" | Mod -> "%"
      | BeatAdd -> "$+" | BeatSub -> "$-" | BeatMul -> "$*" | BeatDiv -> "$/"
      | PCAdd -> "%+" | PCSub -> "%-"
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
      | BeatLess -> "$<" | BeatLeq -> "$<=" | BeatGreater -> "$>" | BeatGeq -> "$>="
      | And -> "&&" | Or -> "||" | BoolEq -> "=="
      | Concat -> "++" | Cons -> ":" | Trans -> "^^" )
      ^ " " ^ string_of_sexpr e2
  | SPrefix(o, e) ->
```

```
      ( match o with Not -> "!" | Inv -> "~" | Retro -> "<>" )
      ^ " " ^ string_of_sexpr e
  | SIf(e1, e2, e3) -> "if " ^ string_of_sexpr e1 ^ " then " ^ string_of_sexpr e2 ^
    " else " ^ string_of_sexpr e3
  | SBeat(i1, i2) -> string_of_sexpr i1 ^
        let rec repeat n s =
            if n>0 then
                repeat (n-1) ("." ^ s)
            else s in repeat i2 ""
  | SNote(pc, reg, bt) -> " (" ^ string_of_sexpr pc ^ ", " ^ string_of_sexpr reg ^ ")$" ^ (
    string_of_sexpr bt)
  | SList(el) -> "[" ^ (String.concat ", " (List.map string_of_sexpr el)) ^ "]"
  | SChord(el) -> "[" ^ (String.concat ", " (List.map string_of_sexpr el)) ^ "]"
  | SSystem(el) -> "[" ^ (String.concat ", " (List.map string_of_sexpr el)) ^ "]"
  | SCall(fname,args) -> fname ^ " " ^ (String.concat " " (List.map string_of_sfargs args))
  | SLet(decs, exp) -> "let " ^ (String.concat " " (List.map string_of_s_dec decs.decls)) ^
                    " in " ^ string_of_sexpr exp
  | SRandom -> "random"
  | SPrint(e) -> "print " ^ string_of_sexpr e

and string_of_sfargs = function
    SArglit(l) -> string_of_int l
  | SArgbool(b) -> string_of_bool b
  | SArgvar(s) -> s
  | SArgbeat(i1, i2) -> string_of_sexpr i1 ^
        let rec repeat n s =
            if n>0 then
                repeat (n-1) ("." ^ s)
            else s in repeat i2 ""
  | SArgnote(pc, reg, bt) -> " (" ^ string_of_sexpr pc ^ ", " ^ string_of_sexpr reg ^ ")$" ^ (
    string_of_sexpr bt)
  | SArgchord(el) -> "[" ^ (String.concat ", " (List.map string_of_sexpr el)) ^ "]"
  | SArgsystem(el) -> "[" ^ (String.concat ", " (List.map string_of_sexpr el)) ^ "]"

  | SArglist(el) -> "[" ^ (String.concat ", " (List.map string_of_sexpr el)) ^ "]"
  | SArgparens(p) -> "(" ^ (string_of_sexpr p)  ^ ")"

and string_of_s_dec = function
    | SFuncdec(f) -> "SFuncdec: \n\t\t" ^ string_of_s_func_decl f
    | SVardef(i, e) -> "SVardef: \n\t\t" ^ string_of_s_ids i ^ "\n\t" ^ string_of_sexpr e
    | SMain(e) -> "SMain: " ^ string_of_sexpr e

and string_of_s_ids i =
    let str = if (i.pats <> []) then String.concat " " (List.map string_of_patterns i.pats)
              else "" in
    "ID: " ^ i.name ^ " " ^ str ^ " :: " ^ String.concat " -> "
    (List.map string_of_s_type i.v_type) ^ "\n"

and string_of_s_func_decl f =
        f.s_fname ^ " (" ^ String.concat ") ("
        (List.map Ast.string_of_patterns f.s_args) ^ ") :: " ^
        String.concat " -> " (List.map string_of_s_type f.type_sig) ^ " = "
        ^ string_of_sexpr f.s_value ^ "\n" ^ string_of_symbol_table f.scope

and string_of_s_type = function
      Int -> "Int"
    | Bool -> "Bool"
    | Note -> "Note"
    | Beat -> "Beat"
    | Chord -> "Chord"
    | System -> "System"
    | List(t) -> "[" ^ string_of_s_type t ^ "]"
    | Empty -> "[]"
    | Poly(s) -> s
    | Unknown -> "Unknown"
    | Num -> "Num"
    | Still_unknown -> "Still Unknown"

and string_of_symbol_table symtab =
    if symtab.parent = None then "Global Scope: \n\t" ^
        String.concat "\t" (List.map string_of_s_ids symtab.identifiers) ^ "\n"
    else (*(string_of_env p) ^ *)"\tNew Scope: \n\t\t" ^
        String.concat "\t\t" (List.map string_of_s_ids symtab.identifiers) ^"\n\t"


let string_of_s_arg = function
  SArglit(i) -> string_of_int i
  | SArgbool(b) -> string_of_bool b
  | SArgvar(s) -> s
  | SArgbeat(e,i) -> (string_of_sexpr e) ^"^"^string_of_int i
  | SArgnote(e1,e2,e3) -> "("^(string_of_sexpr e1)^","^(string_of_sexpr e2)^")$"^(string_of_sexpr e3)
  | SArgchord(el) -> (string_of_sexpr (SChord(el)))
  | SArgsystem(el) -> (string_of_sexpr (SSystem(el)))
  | SArglist(el) -> (string_of_sexpr (SList(el)))
  | SArgparens(e) -> (string_of_sexpr e)

let string_of_s_program p =
    "Program: \n\t" ^ String.concat "\n\t"
    (List.map string_of_s_dec p.decls) ^ "\n" ^
    string_of_symbol_table p.symtab
```

../../Code/sast.ml

```
open Sast
open Ast
open Util

module StringMap = Map.Make(String)

let rec types_to_s_type = function
      TInt -> Sast.Int
    | TBool -> Sast.Bool
    | TNote -> Sast.Note
    | TBeat -> Sast.Beat
    | TChord -> Sast.Chord
    | TSystem -> Sast.System
    | TList(l) -> Sast.List(types_to_s_type l)
    | TPoly(s) -> Sast.Poly(s)

(* Return a list of equivalent types to v1 *)
let equiv_type v1 = match v1 with
    Sast.Chord -> [Sast.List(Sast.Note); Sast.Chord]
    | Sast.System -> [Sast.List(Sast.List(Sast.Note)); Sast.List(Sast.Chord); Sast.System]
    | x -> [x]


(* Return true if v1 and v2 are different types *)
let rec diff_types v1 v2 = match v1, v2 with
    | Sast.List(x)::t1, Sast.List(y)::t2 -> diff_types (x::t1) (y::t2)
    | x::t1, y::t2 -> if ((List.mem x (equiv_type y)) || (List.mem y (equiv_type x)))
                      then diff_types t1 t2 else true
    | [], [] -> false
    | [], _::_ -> true
    | _::_, [] -> true


(* Check if an int is a valid beat *)
let beat_as_int value = if List.mem value [1;2;4;8;16] then true else false

(* Returns true if two types are just ints, beats, or nested ints or beats wher the number of nestings
    for
  both types is equivalent *)
let rec beats_and_ints ty1 ty2 = match ty2, ty2 with
    Sast.List(t1), Sast.List(t2) -> beats_and_ints t1 t2
   | Sast.Beat, Sast.Int -> true
   | Sast.Int, Sast.Beat -> true
   | Sast.Int, Sast.Int -> true
   | Sast.Beat, Sast.Beat -> true
   | _, _ -> false


(* Return true if argument is a system type or a nested system *)
let rec eventual ty = function
    Sast.System | Sast.List(Sast.Chord) | Sast.List(Sast.List(Sast.Note)) ->
        ty = "system"
   | Sast.Beat -> ty = "beat"
   | Sast.Int -> ty = "int"
   | Sast.Unknown -> ty = "unknown"
   | Sast.Empty -> ty = "empty"
   | Sast.List(x) -> if (match ty with
                      "system" -> List.mem x (equiv_type Sast.System)
                    | "beat" -> x = Sast.Beat
                    | "int" -> x = Sast.Int
                    | "unknown" -> x = Sast.Unknown
                    | "empty" -> x = Sast.Empty
                    | _ -> true)
                      then true else eventual ty x
   | _ -> false

(* Check if a type signatures exists for an id in the current scope *)
let rec exists_typesig id = function
```

```
    [] -> false
    | sym_entry :: rest -> if sym_entry.name = id then
                             if sym_entry.v_type <> [Unknown] then true
                             else false
                            else exists_typesig id rest

(* Get the type signature for an identifier in the current scope *)
let get_typesig id ids = (List.find (fun t -> t.name = id) ids).v_type

(* Get type signature for function id in current or higher scope *)
let rec get_types_p id symtab =
    if exists_typesig id symtab.identifiers then get_typesig id symtab.identifiers
    else match symtab.parent with
        | Some(psym) -> get_types_p id psym
        | None -> raise (No_type_signature_found id)

(* Check if a vardef or funcdec exists for an id in the current scope *)
let rec exists_dec id ty = function
    [] -> false
    | SVardef(x, _) :: rest -> if x.name = id then true else exists_dec id ty rest
    | SFuncdec(f) :: rest -> (match ty with
                              "func" -> exists_dec id ty rest
                              | _ -> if f.s_fname = id then true else exists_dec id ty rest)
    | _ :: rest -> exists_dec id ty rest

(* Only checks current scope (might not be needed) *)
let is_declared_here id symtab = List.exists (fun v -> v.name = id) symtab.identifiers

(* checks all scopes if id has been declared *)
let rec is_declared id symtab =
    try
        List.exists (fun v -> v.name = id) symtab.identifiers
    with Not_found ->
        match symtab.parent with
            Some(parent) -> is_declared id parent
        |   _ -> false

(* Add new entry into symbol table or modify existing one if necessary (First Pass work) *)
let mod_var entry symtab =
    if is_declared_here entry.name symtab then
        let preventries = List.filter (fun v -> v.name = entry.name) symtab.identifiers in
        let newsym = List.filter (fun v -> v.name <> entry.name) symtab.identifiers in
        let firsten = List.hd preventries in
        match entry with
        (* Entry is type signature *)
        {v_expr = None } ->
            if List.length preventries = 1 then
                let newen = {name = entry.name; pats = firsten.pats; v_type = entry.v_type;
                             v_expr = firsten.v_expr} in newen :: newsym
            else let newens = List.map (fun en -> let result = {name = en.name; pats = en.pats;
                                                                v_type = entry.v_type;
                                                                v_expr = en.v_expr} in result)
                                        preventries in newens @ newsym
        (* Entry is vardef *)
        | {pats = [] } ->
            let newen = {name = entry.name; pats = entry.pats; v_type = firsten.v_type;
                         v_expr = entry.v_expr} in newen :: newsym
        (* Entry is funcdec *)
        | _ ->
            let newen = {name = entry.name; pats = entry.pats; v_type = firsten.v_type;
                             v_expr = entry.v_expr} in
            if List.length preventries = 1 && firsten.v_expr = None then newen :: newsym
            else newen :: symtab.identifiers
    else entry :: symtab.identifiers

(* Update type of variable definition in our symbol table and our list of declarations *)
let replace_vardef program var oldvar = match var with
    | SVardef(ids, s_expr) ->
        let newdecls = List.filter (fun dec -> dec != oldvar) program.decls in
        let newsym = List.filter (fun v -> v.name <> ids.name) program.symtab.identifiers in
        let newentry = {name = ids.name; pats = []; v_type = ids.v_type; v_expr = ids.v_expr} in
        program.symtab.identifiers <- newentry :: newsym;
        program.decls <- (var :: newdecls); program
    | _ -> program



(* program->string->s_func_decl *)
let rec find_f_def program f_name =
  let decl = List.filter
    (fun dec ->
      match dec with SFuncdec(x)-> x.s_fname = f_name | _ -> false)
      program.decls  in decl
      (*with Not_found ->  raise (Function_not_defined f_name) in
        match  decl with
          SFuncdec(x) -> x
          | _ -> raise (Function_not_defined f_name)
      *)

(* Update type and scope of function declaration in our symbol table and our list of declarations *)
let replace_funcdec program func oldfunc = match func with
    | SFuncdec(info) ->
        let newdecls = List.filter (fun dec -> dec != oldfunc) program.decls in
        let newsym = List.filter (fun v -> v.name <> info.s_fname || v.pats <> info.s_args)
                     program.symtab.identifiers in
        let newentry = {name = info.s_fname; v_type = info.type_sig;
                        pats = info.s_args; v_expr = Some(info.s_value)} in
        program.symtab.identifiers <- newentry :: newsym;
        program.decls <- (func :: newdecls); program
    | _ -> program

let replace_main program new_main =
  let newsym = List.filter (fun v -> v.name <> new_main.name) program.symtab.identifiers in
  program.symtab.identifiers <- new_main :: newsym;
  program

(* Start with an empty symbol table, except for library functions *)
let print_var = { name="print";
                  pats = [Patvar("x")];
                  v_type = [Poly("a"); Poly("a")];
                  v_expr = Some(SPrint(SVariable("x")))}
let random_var = { name = "random";
                   pats = [];
                   v_type = [Int];
                   v_expr = Some(SRandom) }
let global_env = { identifiers = [print_var; random_var]; parent = None }

(* So far, just used to check for pattern errors in collect_pat_vars *)
let rec get_pat_type = function
    Patconst(_) -> Sast.Int
    | Patbool(_)-> Sast.Bool
    | Patvar(_)| Patwild -> Sast.Unknown
    | Patcomma l -> if l = [] then Sast.List(Empty)
                    else let hd = List.hd l in
                     let match_type_or_fail x y =
                        let tx = (get_pat_type x) in
                        let ty = (get_pat_type y) in
                        if tx <> ty && tx <> Sast.Unknown && ty <> Sast.Unknown then
                            raise (Pattern_list_type_mismatch
                                  (string_of_s_type tx ^ " doesn't match " ^ string_of_s_type ty))
                        else () in List.iter (match_type_or_fail hd) l; Sast.List(get_pat_type hd)
    | Patcons (e1, e2) ->
        let ty1 = get_pat_type e1 in
        let ty2 = get_pat_type e2 in
        (match ty2 with
            Sast.Unknown -> Sast.List(ty1)
            | Sast.List(els) -> if eventual "empty" els then Sast.List(ty1)
                                else if ty1 <> els && ty1 <> Sast.Unknown && els <> Sast.Unknown
                                     then raise (Pattern_list_type_mismatch (string_of_s_type ty1
                                                ^ " doesn't match " ^ string_of_s_type els))
                                else if ty1 <> Sast.Unknown then Sast.List(ty1)
                                else Sast.List(els)
    | _ -> raise (Cons_pattern_type_mismatch (string_of_patterns e2)))



(* Collect Variables in pattern *)
let rec collect_pat_vars = function
    [] -> []
    | Patvar(s) :: rest -> s :: collect_pat_vars rest
    | (Patcomma(pl) as l) :: rest -> (match (get_pat_type l) with _ ->  collect_pat_vars pl)
                                     @ collect_pat_vars rest
    | (Patcons(pl1, pl2) as c) :: rest -> (match (get_pat_type c) with _ ->
                                          ((collect_pat_vars [pl1]) @ (collect_pat_vars [pl2])))
                                          @ collect_pat_vars rest
    | _ :: rest -> collect_pat_vars rest

(* Check if there exist 2 function declarations with the same ids and pattern lists *)
let rec same_pats func = function
    [] -> false
    | SFuncdec(info) :: rest ->
        if (info.s_fname <> func.s_fname) then same_pats func rest
        else if (List.length info.s_args <> List.length func.s_args) then same_pats func rest
        else           let rec compare_pats arg1 arg2 = match arg1, arg2 with
            | Patconst(x), Patconst(y) -> if x <> y then false else true
            | Patbool(x), Patbool(y) -> if x <> y then false else true
            | Patvar(_), Patvar(_) -> true
            | Patwild, Patwild -> true
            | Patcomma(l1), Patcomma(l2) ->
                if (List.length l1 <> List.length l2) then false else
                if (List.length l1 = 0 && List.length l2 = 0) then true else
                if (List.length l1 = 0 || List.length l2 = 0) then false else
                if (List.for_all (fun v -> v = true) (List.map2 compare_pats l1 l2))
                then true else false
            | Patcons(p1, p2), Patcons(p3, p4) ->
                if (compare_pats p1 p3 && compare_pats p2 p4) then true else false
            | Patcomma(l1), Patcons(p1, p2) | Patcons(p1, p2), Patcomma(l1) ->
                if (List.length l1 = 0) then false else
                if (compare_pats (List.hd l1) p1) then compare_pats (Patcomma(List.tl l1)) p2
                else false
            | _, _ -> false
            in let result = List.map2 compare_pats info.s_args func.s_args in
                List.for_all (fun v -> v = true) result
    | _ :: rest -> same_pats func rest


(* Set up a new scope given a set of variables to put into scope *)
let rec gen_new_scope = function
    [] -> []
    | pat :: rest -> if List.exists (fun p -> p = pat) rest then raise (Multiple_patterns pat)
                     else {name = pat; pats = []; v_type = [Unknown];
                           v_expr = None} :: gen_new_scope rest

let rec find_var_entry symtab v =
  try ( List.find (fun t -> t.name = v) symtab.identifiers)
    with Not_found ->
        (match symtab.parent with
      Some(p) -> find_var_entry p v
      | None -> raise (Missing_variable_definition ("find_var"^v)))

let rec find_func_entry symtab f =
  let func_list = List.filter (fun t -> t.name = f) symtab.identifiers
    in if (List.length func_list) >0 then func_list
    else (match symtab.parent with
      Some(p) -> find_func_entry p f
      | None -> raise (Function_not_defined f))

let change_type symtab old_var n_type =
  let new_var = {name = old_var.name;
                pats = old_var.pats;
                v_type = [n_type];
                v_expr = old_var.v_expr} in
  let other_vars = List.filter
    (fun vs -> vs.name <> old_var.name)
    symtab.identifiers in
  { parent = symtab.parent; identifiers = new_var :: other_vars}


let rec check_type_equality t1 t2 =
match t1 with
    Sast.Chord -> (match t2 with
      Sast.List(b) ->  b = Sast.Note
    | Sast.Chord -> true
    | Unknown -> true
    | _ -> false )
  | Sast.System -> (match t2 with
      Sast.List(b) -> check_type_equality b Sast.Chord
    | Sast.System -> true
    | Unknown -> true
    | _ -> false )
  | Sast.List(a) -> (match t2 with
      Sast.List(b) -> check_type_equality a b
    | Sast.Empty -> true
    | Unknown -> true
    | _ -> false )
  | Sast.Empty -> (match t2 with
      Sast.List(b) -> true
    | Sast.Empty -> true
    | Unknown -> true
    | _ -> false )
  | Sast.Poly(a) ->  true (* shouldn't be used with poly types *)
  | Sast.Unknown -> true (* should only be used with known types *)
  | Sast.Still_unknown -> raise (Type_error "having trouble resolving types")
  | Sast.Int -> ( match t2 with
      Sast.Int -> true
    | Sast.Unknown -> true
    | Sast.Poly(b) -> true
    | Sast.Beat -> true
    | _ -> false)
  | Sast.Beat -> ( match t2 with
      Sast.Beat -> true
    | Sast.Unknown -> true
    | Sast.Poly(b) -> true
    | Sast.Int -> true
    | _ -> false)
  | _ -> (match t2 with
      Sast.Poly(b) -> true (* shouldn't be used with poly types *)
    | Sast.Unknown -> true (* should only be used with known types *)
    | Sast.Still_unknown -> raise (Type_error "having trouble resolving types")
    | _ -> t1 = t2 )

let rec try_get_type pm ts tr = match ts with
    Sast.Poly(a) -> if StringMap.mem a pm then StringMap.find a pm
            else if(tr = Unknown) then ts else tr
  | Sast.List(a) -> (match tr with
      Sast.List(b) -> Sast.List(try_get_type pm a b)
    | _ -> if (tr = Unknown ) then ts else tr)
  | _ -> ts

(* Returns a type from an expression*)
let rec get_type short symtab = function
      SLiteral(l) -> Int
    | SBoolean(b) -> Bool
    | SVariable(s) ->
      let var = find_var_entry symtab s in
        let ts = var.v_type in
        if(List.length ts <> 1) then raise (Function_used_as_variable s)
        else let t = List.hd ts in
          if(t <> Unknown) then t
          else
            (match var.v_expr with
            Some(expr) ->
              let symtab = (change_type symtab var Still_unknown) in
              get_type short  symtab expr
            | None -> if(short) then Sast.Unknown else (raise (Missing_variable_definition ("SVariable "^
 s))))
    | SBinop(e1, o, e2) ->  (* Check type of operator *)
        let te1 = get_type short symtab e1
        and te2 = get_type short symtab e2 in
            (match o with
                Ast.Add | Ast.Sub | Ast.Mul | Ast.Div | Ast. Mod |
                Ast.PCAdd | Ast.PCSub ->
                (* Arithmetic Operators *)
                if(short) then Sast.Int
                else
                    if te1 <> Sast.Int && (match te1 with Poly(_) -> false | _ -> true)
                    then type_error ("First element of an arithmetic binary operation " ^
                        "must be of type Int but element was of type " ^
                        Sast.string_of_s_type te1)
                    else
                        if te2 <> Sast.Int && (match te1 with Poly(_) -> false | _ -> true)
                        then type_error ("Second element of an arithmetic binary operation " ^
                            "must be of type Int but element was of type " ^
                            Sast.string_of_s_type te2)
                        else Sast.Int
                | Ast.Less | Ast.Leq | Ast.Greater | Ast.Geq ->
                  (* Comparison Operators *)
                 if(short) then Sast.Bool
                 else
                    if te1 <> Sast.Int
                    then type_error ("First element of a comparison binary operation " ^
                        "must be of type Int but element was of type " ^
                        Sast.string_of_s_type te1)
                    else
                        if te2 <> Sast.Int
                        then type_error ("Second element of a comparison binary operation " ^
                            "must be of type Int but element was of type " ^
                            Sast.string_of_s_type te2)
                        else Sast.Bool
                | Ast.BeatAdd | Ast.BeatSub | Ast.BeatDiv | Ast.BeatMul ->
                  (* Beat Arithmetic Operators *)
                if(short) then Sast.Beat
                else
                    if te1 <> Sast.Int && te1 <> Sast.Beat
```

```
                     then type_error ("First element of a Beat arithmetic binary " ^
                         "operation must be of types Int or Beat but element was of type " ^
                         Sast.string_of_s_type te1)
                     else
                         if te2 <> Sast.Int && te2 <> Sast.Beat
                         then type_error ("Second element of a Beat arithmetic binary " ^
                             "operation must be of types Int or Beat but element was of type " ^
                             Sast.string_of_s_type te2)
                         else Sast.Beat
                 | Ast.BeatLess | Ast.BeatLeq | Ast.BeatGreater | Ast.BeatGeq ->
                   (* Beat Comparison Operators *)
                  if(short) then Sast.Bool
                  else
                     if te1 <> Sast.Int && te1 <> Sast.Beat
                     then type_error ("First element of a Beat comparison binary " ^
                         "operation must be of types Int or Beat but element was of type " ^
                         Sast.string_of_s_type te1)
                     else
                         if te2 <> Sast.Int && te2 <> Sast.Beat
                         then type_error ("Second element of a Beat comaprison binary " ^
                             "operation must be of types Int or Beat but element was of type " ^
                             Sast.string_of_s_type te2)
                         else Sast.Bool
                 | Ast.And | Ast.Or ->  (* Boolean Operators: Bool && Bool, Bool || Bool *)
                  if(short) then Sast.Bool
                  else
                     if te1 <> Sast.Bool
                     then type_error ("First element of a boolean binary operation " ^
                         "must be of type Bool but element was of type " ^
                         Sast.string_of_s_type te1)
                     else
                         if te2 <> Sast.Bool
                         then type_error ("Second element of a boolean binary operation " ^
                             "must be of type Bool but element was of type " ^
                             Sast.string_of_s_type te2)
                         else Sast.Bool
                 | Ast.BoolEq -> (* Structural Comparision: Element == Element *)
                  if(short) then Sast.Bool
                 else
                     if te1 <> te2 && (match te1, te2 with Poly(_), _ | _, Poly(_) -> false | _ -> true)
                     then type_error ("Elements must be of same type for " ^
                         "structural comparison. First element has type " ^
                         Sast.string_of_s_type te1 ^ " and second element has type " ^
                         Sast.string_of_s_type te2)
                     else Sast.Bool
                 | Ast.Concat -> (* Concat: List ++ List *)
                 if(short) then Sast.Empty (* fix *)
                 else
                     (* Not sure this checks the correct thing *)
                     (match te1 with
                       Sast.List(t1) -> (match te2 with
                           Sast.List(t2) -> if t1 <> t2 then
                               (try
                                 let x = get_type short symtab (SList([e1;e2])) in
                                 (fun v -> match v with Sast.List(x) -> x | _ -> type_error("PROBLEM")) x
                               with (Type_error x) ->
                                   type_error ("Operands of a concat operator have different types"))
                               else te1
                         | Sast.Empty -> te1
                         | _ -> type_error "Concat operator can only used between lists")
                     | Sast.Chord -> (match te2 with
                           Sast.Chord | Sast.Empty | Sast.List(Sast.Note) -> Sast.Chord
                         | _ -> type_error ("Operands of a concat operator have different types"))
                     | Sast.System -> (match te2 with
                           Sast.System | Sast.List(Sast.Chord) | Sast.List(Sast.List(Sast.Note))
                           | Sast.Empty -> Sast.System
                         | _ -> type_error ("Operands of a concat operator have different types"))
                     | Sast.Empty -> (match te2 with
                           Sast.List(t2) -> te2
                         | Sast.Empty -> Sast.Empty
                         | Sast.Chord -> Sast.Chord
                         | Sast.System -> Sast.System
                         | _ -> type_error "Concat operator can only used between lists")
                     | _ -> type_error "Concat operator can only used between lists")

                 | Ast.Cons -> (* Cons: Element : List *)
                 if(short) then Sast.Empty (* ? *)
                 else
                     (match te2 with
                        Sast.List(t2) -> (if diff_types [te1] [t2] && te1 <> Sast.Empty then
                               (try
                                 let x = get_type short symtab (SList([e1;e2])) in
                                 (match e2 with
                                     SCall(_,_) -> x
                                   | _ -> (fun v -> match v with Sast.List(x) -> x | _ -> type_error("
PROBLEM")) x)
                               with (Type_error x) ->
                                   type_error (x))
                             else te2)
                      | Sast.Chord -> (if te1 <> Sast.Empty && te1 <> Sast.Note && te1 <> Sast.Empty then
                          type_error ("The types of the lhs and rhs of a cons operator don't match")
                          else te2)
                      | Sast.System -> (if te1 <> Sast.Empty && te1 <> Sast.Chord && te1 <> Sast.List(Sast
.Note)  then
                          type_error ("The types of the lhs and rhs of a cons operator don't match")
                          else te2)
                      | Sast.Empty -> (match te1 with
                            Sast.Note -> Sast.Chord
                          | Sast.Chord -> Sast.System
                          | _ -> Sast.List(te1))
                      | _ -> type_error ("The second operand of a cons operator was: "
                          ^ (Sast.string_of_s_type te2) ^ ", but a type of list was expected"))
                 | Ast.Trans -> (* Trans: Int ^^ List *)
                 if(short) then Sast.List(Sast.Int)
                 else
                     if te1 <> Sast.Int
                     then type_error ("First element in a Trans expression " ^
                         "must be of type Int but element was of type " ^
                         Sast.string_of_s_type te1)
                     else
                         if te2 <> Sast.List(Sast.Int)
                         then type_error ("Second element in a Trans expression " ^
                             "must be a List of type Int but element was of type " ^
                             Sast.string_of_s_type te2)
                         else te2
             )
    | SPrefix(o, e) -> (* Prefix Operators *)
        let te = get_type short symtab e in
        (match o with
            Ast.Not -> (* Not: ! Bool *)
                if te <> Sast.Bool
                then type_error ("Element in Not operation must be of type Bool " ^
                    "but element was of type " ^ Sast.string_of_s_type te)
                else te
            | Ast.Inv | Ast.Retro -> (* Row Inversion: ~ List, Row Retrograde: <> List*)
                if te <> Sast.List(Sast.Int)
                then type_error ("Element in Prefix operation must be a List of " ^
                    "type Int but element was of type " ^ Sast.string_of_s_type te)
                else te
        )
    | SIf(e1, e2, e3) -> (* Check both e2 and e3 and make sure the same *)
        let te1 = get_type short symtab e1 in
        if te1 <> Sast.Bool then
            type_error (string_of_sexpr e1 ^ " has type " ^ string_of_s_type te1
            ^ " but is used as if it has type " ^ string_of_s_type Sast.Bool)
        else let te2 = get_type short symtab e2 in
             let te3 = get_type short symtab e3 in
             if te2 <> te3 && (match te2, te3 with Sast.Empty, Sast.List(_) |
                                                   Sast.List(_), Sast.Empty -> false
                                                   | _, _ -> true) then
                type_error (string_of_sexpr e2 ^ " has type " ^ string_of_s_type te2
                ^ " but " ^ string_of_sexpr e3 ^ " has type " ^ string_of_s_type te3
                ^ " which is not allowed in conditional statement")
                else te2
    | SBeat(i1, i2) ->
        let ti1 = get_type short symtab i1 in
        if ti1 <> Sast.Int
        then type_error ("First element in a Beat must be of type Int " ^
            "and a power of 2 between 1 and 16. The given element was of type " ^
            Sast.string_of_s_type ti1)
        else
          (* Checked more thoroughly in interpreter *)
          if i2 < 0 || i2 > 4
            then type_error ("Dots may not increase Beat value past 16th")
            else Sast.Beat
    | SNote(pc, reg, b) ->
        let tpc = get_type short symtab pc
        and treg = get_type short symtab reg
        and tb = get_type short symtab b in
        if tpc <> Sast.Int
        then type_error ("First element in Note (pitch class) must be of type Int " ^
            "between -1 and 11 but element was of type " ^ Sast.string_of_s_type tpc)
        else
            if treg <> Sast.Int
            then type_error ("Second element in Note (register) must be of type Int " ^
                "between 0 and 3 but element was of type " ^ Sast.string_of_s_type tpc)
            else
                if tb <> Sast.Int && tb <> Sast.Beat
                then type_error ("Third element in Note (Beat) must be of type Beat " ^
                    "but element was of type " ^ Sast.string_of_s_type tb)
                else Sast.Note
    | SList(el) -> (* Check all elements have same type*)
        (match el with
          [] -> Sast.Empty
        | _ -> let hd = List.hd el in
             let match_type_or_fail x y =
                let tx = (get_type short symtab x) in
                let ty = (get_type short symtab y) in
                if diff_types [tx] [ty] && (not (beats_and_ints tx ty) || not (contains_beat symtab el))
                    then type_error (string_of_sexpr x ^ " has type of "
                        ^ Sast.string_of_s_type tx ^ " but "
                        ^ string_of_sexpr y ^ " has type "
                        ^ Sast.string_of_s_type ty ^ " in a same list")
                else ()
            in List.iter (match_type_or_fail hd) el;
            if contains_beat symtab el then Sast.List(powers_of_two symtab el)
            else Sast.List(get_type short symtab (hd)))
    | SChord(el) -> (* Check all elements have type of TNote *)
        let hd = List.hd el in
            let match_type_or_fail x y =
                let tx = (get_type short symtab x) in
                let ty = (get_type short symtab y) in
                if tx <> ty
                    then type_error ("Elements in Chord should all have type of "
                    ^ Ast.string_of_types Ast.TNote ^ " but the element of "
                    ^ string_of_sexpr y ^ " has type of " ^ Sast.string_of_s_type ty)
                else () in List.iter (match_type_or_fail hd) el;
        let hd = List.hd el in
            let match_duration_or_fail x y = match x, y with
                 SNote(p1,r1,bt1), SNote(p2,r2,bt2) ->
                    (if (string_of_sexpr bt1) <> (string_of_sexpr bt2)
                        then type_error ("The time durating of " ^ string_of_sexpr bt1
                        ^ " is not the consistent with that of " ^ string_of_sexpr bt2)
                        else ())
                | _,_ -> type_error ("Not Expected Exception")
        in List.iter (match_duration_or_fail hd) el; Sast.Chord
    | SSystem(el) -> (* Check all elements have type of TChord *)
        let hd = List.hd el in
            let match_type_or_fail x y =
                let tx = (get_type short symtab x) in
                let ty = (get_type short symtab y) in
                if tx <> ty
                    then type_error ("Elements in Chord should all have type of "
                    ^ string_of_s_type Sast.Chord ^ " but the element of "
                    ^ string_of_sexpr y ^ " has type of " ^ string_of_s_type ty)
                else () in List.iter (match_type_or_fail hd) el; Sast.System
    | SLet(decs, exp) -> get_type short decs.symtab exp
    | SRandom -> Sast.Int
    | SPrint(e) -> get_type short symtab e
    | SCall(f, args) ->
            if(short) then let f_vars = find_func_entry symtab f in
                try (List.hd (List.rev ((List.hd f_vars).v_type)))
                    with _ -> Unknown
            else
      let poly_map = StringMap.empty in
        let f_vars = find_func_entry symtab f in
        let f_entrys = match_args symtab [] f_vars args in
        let f_entry = if(List.length f_entrys)>0 then
          if (List.length f_entrys) = 1 then List.hd f_entrys
          else (let st = try
          List.find (fun t -> (List.length t.v_type)>0) f_entrys with
            _ ->raise (Type_error ("function not found " ^ f)) in
            {name = st.name; pats = []; v_type = st.v_type; v_expr=None})
                 (*(try List.find (fun t ->
                    has_pattern (Patconst(0)) t.pats||
                    has_pattern (Patbool(true)) t.pats||
                    has_pattern (Patcomma([])) t.pats) f_entrys with _ ->
                    (try List.find (fun t ->
                    has_pattern (Patcomma([Patconst(0)])) t.pats ||
                    has_pattern (Patcons(Patconst(0),Patconst(0))) t.pats ) f_entrys with _ ->
                        (try List.find (fun t ->
                        has_pattern (Patvar("a")) t.pats) f_entrys with _ ->
                            (try List.find(fun t ->
                             has_pattern Patwild t.pats) f_entrys with _ ->
                                 raise (Type_error ("you have to have some pattern")))))) *)
                 else raise (Type_error ("function not found " ^ f))
                      in
         let ts_id = try List.find (fun t-> (List.length t.v_type)>0) f_entrys with
           _ -> raise (Type_error ("function not found " ^ f)) in
         let tsig = List.hd (List.rev ts_id.v_type) in
         let pm = StringMap.add "print" Unknown poly_map in
         let return_type = (match f_entry.v_expr with
           Some(e) -> if not (is_recursive f e) then (
                     try(get_type false symtab e) with _-> Unknown)
                     else Unknown | None -> Unknown ) in
         let polymap = map_return f pm
                        tsig
                         return_type in
         let full_map = check_arg_types f symtab polymap args f_entry.v_type in
         try_get_type  full_map tsig return_type
       (* check all args against f type sig *)
       (* check expr matches last type *)

and is_recursive func = function
    SBeat(e,i) -> is_recursive func e
    | SNote(e1,e2,e3) -> is_recursive func e1 || is_recursive func e2 || is_recursive func e3
    | SBinop(e1, op, e2) -> is_recursive func e1 || is_recursive func e2
    | SPrefix(op, e) -> is_recursive func e
    | SIf(e1,e2,e3) -> is_recursive func e1 || is_recursive func e2 || is_recursive func e3
    | SList(elist)
    | SSystem(elist)
    | SChord(elist) -> List.fold_left (||) false (List.map (is_recursive func) elist)
    | SCall(f, args) -> let b = f = func in b
    | SLet(p, e) -> is_recursive func e
    | SPrint(e) -> is_recursive func e
    | _ -> false

and has_pattern pat pat_list =
    List.fold_left (||) false (List.map (fun p -> match p with
        Patconst(i) -> (match pat with
            Patconst(i2) -> true
          | _ -> false)
      | Patbool(b) -> (match pat with
            Patbool(b2) -> true
          | _ -> false)
      | Patvar(v) -> (match pat with
            Patvar(v2) -> true
          | _ -> false)
      | Patwild -> (match pat with
            Patwild -> true
          | _ -> false)
      | Patcomma(l) -> (match pat with
            Patcomma([])-> l = []
          | Patcomma(l) -> l <> []
          | _ -> false)
      | Patcons(p1,p2) -> (match pat with
            Patcons(p3,p4) -> true
          | _ -> false)) pat_list)

and map_return f  pm ts ret = match ts with
    Sast.Poly(a) -> (match ret with
          Unknown -> pm  (* is argument to function? *)
        | Still_unknown -> pm
        | Sast.Poly(b) -> map_return f pm ret ret
        | _ -> StringMap.add a ret pm)
  | _ ->
        if check_type_equality ts ret
      then pm
      else type_error ("Mismatch return type "^f)

and get_arg_type f prog a = match a with
    SArglit(i) -> Sast.Int
  | SArgbool(b) -> Sast.Bool
  | SArgvar(v) -> (try(get_type false prog (SVariable(v))) with _-> Sast.Unknown)
  | SArgbeat(e,i) -> Sast.Beat
  | SArgnote(e1,e2,e3) -> Sast.Note
  | SArgchord(elist) -> Sast.Chord
  | SArgsystem(elist) -> Sast.System
  | SArglist(elist) -> get_type false prog (SList(elist))
  | SArgparens(e) ->  try (get_type true prog e)
        with _ -> Sast.Unknown


and map_args_with_t name poly_map (a_t, t) =
    match t with
       Poly(t_n) -> if StringMap.mem t_n poly_map then
            let typ = StringMap.find t_n poly_map in
                if(check_type_equality typ a_t)
                then poly_map
                else raise (Function_arguments_type_mismatch ("1."^name^" "^(string_of_s_type t)) )
            else StringMap.add t_n a_t poly_map
    | Sast.List(l) -> (match a_t with
        Sast.List(lt) -> map_args_with_t name poly_map (lt, l)
      | Sast.Chord -> map_args_with_t name poly_map (Sast.Note, l)
      | Sast.System -> map_args_with_t name poly_map (Sast.Chord, l)
      | Sast.Empty -> poly_map
      | _ -> raise (Function_arguments_type_mismatch  ("2."^name^" "^(string_of_s_type t)^ " "^(
    string_of_s_type a_t))))
    | _ -> if check_type_equality t a_t then poly_map
        else raise (Function_arguments_type_mismatch  ("3."^name^" "^(string_of_s_type t)^" "^(
    string_of_s_type a_t)))

and map_args name prog poly_map (a,t) =
   match t with
    Poly(t_n) -> if StringMap.mem t_n poly_map then
        let typ = StringMap.find t_n poly_map in
          if(check_type_equality typ (get_arg_type name prog a))
          then poly_map
          else raise (Function_arguments_type_mismatch (name ^ " "^(string_of_s_arg a)))
          (* check types *)
        else StringMap.add t_n (get_arg_type name prog a) poly_map
    | Sast.List(l) -> (match a with
          SArglit(i) -> raise (Function_arguments_type_mismatch (name ^ " "^(string_of_s_arg a)))
        | SArgbool(b) -> raise (Function_arguments_type_mismatch (name ^ " "^(string_of_s_arg a)))
        | SArglist(e) ->let typ = get_arg_type name prog a in
            if(typ = Unknown) then poly_map
            else( match typ with
                Sast.List(lt) -> map_args_with_t name poly_map (lt, l)
              | Sast.Chord -> map_args_with_t name poly_map (Sast.Note, l)
              | Sast.System -> map_args_with_t name poly_map (Sast.Chord, l)
              | Sast.Empty -> poly_map
              | _ -> poly_map)
        | SArgparens(e) ->let typ = get_arg_type name prog a in
            if(typ = Unknown) then poly_map
            else( match typ with
                Sast.List(lt) -> map_args_with_t name poly_map (lt, l)
              | Sast.Chord -> map_args_with_t name poly_map (Sast.Note, l)
              | Sast.System -> map_args_with_t name poly_map (Sast.Chord, l)
              | Sast.Empty -> poly_map
              | _ -> poly_map)
        | SArgvar(e) -> let typ = get_arg_type name prog a in
            if(typ = Unknown) then poly_map
            else( match typ with
                Sast.List(lt) -> map_args_with_t name poly_map (lt, l)
              | Sast.Chord -> map_args_with_t name poly_map (Sast.Note, l)
              | Sast.System -> map_args_with_t name poly_map (Sast.Chord, l)
              | Sast.Empty -> poly_map
              | _ -> poly_map)
        | SArgchord(elist) -> map_args_with_t name poly_map(Sast.Note, l)
        | SArgsystem(elist) -> map_args_with_t name poly_map(Sast.Chord, l)
        | _ -> raise (Function_arguments_type_mismatch ("List "^name^ " "^(string_of_s_arg a))))
  | _ ->
      if check_type_equality t  (get_arg_type name prog a) then poly_map
      else raise (Function_arguments_type_mismatch ("Other "^name ^ " "^(string_of_s_arg a)))


(* If an Int is in the given list of s_exprs, make sure it's a power of two and return Beat type if so *)
and powers_of_two program = function
    | [] -> Sast.Beat
    | SList(sexpr) :: rest -> Sast.List(powers_of_two program (sexpr @
                                            (let rec delist = function
                                             [] -> []
                                             |SList(sexpr)::r -> sexpr @ delist r
                                             |SVariable(s)::r -> delist r (* Ignoring vars...resolve this
     in interp! *)
                                             |_ -> type_error ("Found a list of nested elements
                                                                 with non-equal number of nestings")
                                             in delist rest)))
    | SLiteral(i) :: rest -> if beat_as_int i then powers_of_two program rest else
                                type_error ("Non-power of 2 entity " ^ (string_of_int i) ^
                                            " in list of beat elements")
    | x :: rest -> let tyx = get_type false program x in (match tyx with
                      Sast.Beat | Sast.Int -> powers_of_two program rest
                     | y  -> if eventual "beat" tyx || eventual "int" tyx then powers_of_two program rest
                             else type_error ("Element in list of beats and/or ints is neither a beat
                                                  nor an int " ^ (string_of_sexpr x)))

(* Check if we have a Beat expression in a list of s_exprs *)
and contains_beat program = function
    [] -> false
    | SList(sexpr)::rest -> if contains_beat program sexpr then true else contains_beat program rest
    | SBeat(_,_)::rest -> true
    | x :: rest -> if eventual "beat" (get_type false program x) then true else contains_beat program
    rest


and check_arg_types name prog poly_map a_list t_list =
  if((List.length a_list) +1) <> (List.length t_list) then
    raise (Wrong_number_of_arguments name)
  else let t_list = List.rev (List.tl (List.rev t_list)) in
        let a_list = List.rev a_list in
    let tup = List.combine a_list t_list in
      let poly_map = (List.fold_left (map_args name prog) poly_map tup) in poly_map

and match_pat_expr pat e_t =
match pat with
  Patconst(i1) -> (match e_t with
      Sast.Int -> true
        | Unknown -> true
        | Sast.Still_unknown -> true
        | Sast.Poly(a) -> true
    | _ -> false)
  |Patbool(b1) -> (match e_t with
      Sast.Bool -> true
        | Unknown -> true
        | Sast.Still_unknown -> true
        | Sast.Poly(a) -> true
    | _ -> false)
  |Patvar(s) -> true
  |Patwild -> true
  |Patcomma(pl) -> (match e_t with
      Sast.List(lt) -> if List.length pl > 0
                  then match_pat_expr (List.hd pl) lt
                  else false
    | Sast.Chord -> if List.length pl > 0
                  then match_pat_expr (List.hd pl) Sast.Note
                  else false
    | Sast.System -> if List.length pl > 0
                  then match_pat_expr (List.hd pl) Sast.Chord
                  else false
    | Sast.Empty -> if List.length pl = 0 then true else false
        | Sast.Unknown -> true
        | Sast.Still_unknown -> true
        | Sast.Poly(a) -> true
    | _ -> false)
  |Patcons(p1,p2) -> (match e_t with
      Sast.List(lt)->(match_pat_expr p1 lt)&&(match_pat_expr p2 e_t)
    | Sast.Chord->(match_pat_expr p1 Sast.Note)&&(match_pat_expr p2 Sast.Chord)
    | Sast.System->(match_pat_expr p1 Sast.Chord)&&(match_pat_expr p2 Sast.System)
        | Sast.Unknown -> true
        | Sast.Still_unknown -> true
        | Sast.Poly(a) -> true
    | _ -> false)

and match_arg prog (pat, arg) =
match pat with
    Patconst(i1) -> (match arg with
        SArglit(i2) -> i1 = i2
      | SArgvar(s) -> let typ = (try(get_type false prog (SVariable(s))) with _ -> Sast.Unknown )in
        check_type_equality typ Sast.Int
      | SArgparens(e1) -> check_type_equality (get_type false prog e1) Sast.Int
      | _ -> false )
  | Patbool(b1) -> (match arg with
        SArgbool(b2) -> b1 = b2
      | SArgvar(s) -> check_type_equality (try(get_type false prog (SVariable(s))) with _ -> Sast.Unknown
    ) Sast.Bool
      | SArgparens(e1) ->check_type_equality (get_type false prog e1 ) Sast.Bool
      | _ -> false)
  | Patvar(v1) -> true
  | Patwild -> true
  | Patcomma(pat_list) -> (match arg with
        SArgchord(el) -> match_pat_expr pat Sast.Chord
      | SArgsystem(el) -> match_pat_expr pat Sast.System
      | SArglist(el) -> match_pat_expr pat (get_type false prog (SList(el)))
      | SArgparens(s_expr) -> match_pat_expr pat (get_type false prog s_expr)
      | SArgvar(s) -> match_pat_expr pat (get_type false prog (SVariable(s)))
      | _ -> false)
  | Patcons(pat1,pat2) -> (match arg with
      SArglist(el) -> match_pat_expr pat (get_type false prog (SList(el)))
    | SArgchord(el) -> match_pat_expr pat Sast.Chord
    | SArgsystem(el) -> match_pat_expr pat Sast.System
    | SArgparens(e) -> match_pat_expr pat (get_type false prog e)
    | SArgvar(s) -> match_pat_expr pat (get_type false prog (SVariable(s)))
    | _ -> false )


and match_args prog l id_list args = let args = List.rev args in match id_list with
  [] -> l
  |(a::b) ->
    let comb = (try List.combine a.pats args with _ -> []) in
    let is_match = List.fold_left (&&) true
      (List.map (match_arg prog) comb) in
      if(is_match) then a :: (match_args prog l b (List.rev args))
      else match_args prog l b (List.rev args)


let rec type_is_equal t1 t2 =
  if( t1 = t2 ) then true
  else match t1 with
      Sast.List(a) -> (match t2 with
          Sast.List(b) -> type_is_equal a b
                | Sast.Chord -> type_is_equal a Sast.Note
                | Sast.System -> type_is_equal a Sast.Chord
        | Sast.Poly(b) -> true
        | Empty -> true
        | _ -> false )
    | Sast.Poly(a) -> true
    | Sast.Empty -> (match t2 with
          Sast.List(b) -> true
        | _ -> false)
    | _ -> (match t2 with
          Sast.Poly(b) -> true
        | _ -> false )

let check_ret_type symtab types info =
  (* Check that function value has correct type *)
    let typ_sig = (List.hd (List.rev types)) in
    let get_t_typ = (get_type true symtab info.s_value) in
    if not( type_is_equal typ_sig get_t_typ )
    then raise (Type_mismatch ("Expression of function " ^ info.s_fname ^
                    " " ^ String.concat " " (List.map string_of_patterns info.s_args)))
    else symtab.identifiers <- {name = info.s_fname; pats = info.s_args; v_type = info.type_sig; v_expr =
      Some(info.s_value)} :: symtab.identifiers;
              symtab

let rec matching_patterns polypats expected actual = match expected, actual with
   |  ex::rest, act::rest2 -> if ex = act then matching_patterns polypats rest rest2 else
                            (match ex with
                             Poly(id) -> if List.exists (fun (poly,ty) -> poly = id && ty != act) polypats
                                         then false else matching_patterns ((id,act) :: polypats) rest
     rest2
                            | Sast.List(_) -> if (eventual "empty" act) || (eventual "unknown" act) then
     matching_patterns polypats rest rest2
                                               else false
                            | _ -> if eventual "unknown" act then matching_patterns polypats rest rest2
     else false)
   | [], [] -> true
   | _, _ -> false

let rec check_pat_types types info =
      let exp_pattypes = (List.rev (List.tl (List.rev types))) in
         let act_pattypes = (List.map get_pat_type info.s_args) in
         if not (matching_patterns [] exp_pattypes act_pattypes) then
            raise (Type_mismatch ("Patterns don't match type signature for " ^ info.s_fname ^
                    " " ^ String.concat " " (List.map string_of_patterns info.s_args)))
         else let pat_pairs = List.combine info.s_args exp_pattypes in
              let rec gen_scope = function
                  [] -> []
                | (p, ty) :: rest ->
                   (match p, ty with
                    Patvar(s), _ -> {name = s; pats = []; v_type = [ty];
                                     v_expr = None} :: gen_scope rest
                   | Patcomma(l), Sast.List(lty) ->
                       let tups = List.map (fun v -> (v, lty)) l in
                       (gen_scope tups) @ gen_scope rest
                   | Patcomma(l), Sast.Poly(s) ->
                       let tups = List.map (fun v -> (v, Sast.Unknown)) l in
```

```
                         (gen_scope tups) @ gen_scope rest
                     | Patcons(l1,l2), Sast.List(lty) ->
                         (gen_scope [(l1, lty)]) @ (match l2 with
                                                     | Patvar(s) -> [{name = s; pats = [];
                                                                     v_type = [ty];
                                                                     v_expr = None}]
                                                     | _ -> (gen_scope [(l2, ty)]))
                         @ gen_scope rest
                     | Patcons(l1,l2), Sast.Poly(s) ->
                         (gen_scope [(l1, Sast.Unknown)]) @ (match l2 with
                                                     | Patvar(s) -> [{name = s; pats = [];
                                                                     v_type = [ty];
                                                                     v_expr = None}]
                                                     | _ -> (gen_scope [(l2,ty)]))
                     | _ -> gen_scope rest) in
         info.scope.identifiers <- gen_scope pat_pairs;info.scope

let rec main_type_check = function
   Sast.Empty -> true
 | Sast.Note -> true
 | Sast.Chord -> true
 | Sast.System -> true
 | Sast.List(sys) -> main_type_check sys
 | _ -> false


(* First pass walk_decl -> Try to construct a symbol table *)
let rec walk_decl prog = function
    Ast.Tysig(id,types) ->
                let entry = {name=id; pats = []; v_type = (List.map types_to_s_type types);
                            v_expr = None} in
                if (exists_typesig id prog.symtab.identifiers)
                    then raise (Multiple_type_sigs id)
                else prog.symtab.identifiers <- mod_var entry prog.symtab; prog
    | Ast.Vardef(id, expr) ->
                let var = {name=id; pats = []; v_type = [Unknown];
                          v_expr = Some(to_sexpr prog.symtab expr)} in
                if(exists_dec id "var" prog.decls)
                    then raise (Multiple_declarations id)
                else prog.symtab.identifiers <- mod_var var prog.symtab;
                    { decls = SVardef(var, (to_sexpr prog.symtab expr)) :: prog.decls ;
                    symtab = prog.symtab}
    | Ast.Funcdec(fdec) ->
            if (exists_dec fdec.fname "func" prog.decls)
                then raise (Multiple_declarations fdec.fname)
            else
                let f_vars = collect_pat_vars fdec.args in
                let new_scope = {parent=Some(prog.symtab); identifiers = gen_new_scope f_vars} in
                let funcdef = SFuncdec({s_fname = fdec.fname;
                                        type_sig = [Unknown];
                                        s_args = fdec.args;
                                        s_value = to_sexpr new_scope fdec.value;
                                        scope = new_scope;}) in
                let var = {name = fdec.fname; pats = fdec.args;  v_type = [Unknown];
                           v_expr = Some(to_sexpr prog.symtab fdec.value)} in
                    prog.symtab.identifiers <- mod_var var prog.symtab;
                    { decls = funcdef :: prog.decls; symtab = prog.symtab }
    | Main(expr) ->
        if(prog.symtab.parent = None) then
            if( is_declared "main" prog.symtab)
              then raise (Multiple_declarations "main")
            else let mainvar = {name = "main";
                                pats = [];
                                v_type = [Unknown];
                                v_expr = Some(to_sexpr prog.symtab expr)}
              in prog.symtab.identifiers <- (mod_var mainvar prog.symtab);
               { decls = (prog.decls @ [SMain(to_sexpr prog.symtab expr)]); symtab = prog.symtab }
        else raise Main_wrong_scope


(* Convert Ast expression nodes to Sast s_expr nodes (so we can have nested scopes) *)
and to_sexpr symbol = function
    | Ast.Literal(i) -> SLiteral(i)
    | Ast.Boolean(b) -> SBoolean(b)
    | Ast.Variable(s) -> SVariable(s)
    | Ast.Beat(e, i) -> SBeat(to_sexpr symbol e, i)
    | Ast.Note(e1, e2, e3) -> SNote(to_sexpr symbol e1, to_sexpr symbol e2, to_sexpr symbol e3)
    | Ast.Binop(e1, op, e2) -> SBinop(to_sexpr symbol e1, op, to_sexpr symbol e2)
    | Ast.Prefix(pop, e) -> SPrefix(pop, to_sexpr symbol e)
    | Ast.If(e1,e2,e3) -> SIf(to_sexpr symbol e1, to_sexpr symbol e2, to_sexpr symbol e3)
    | Ast.List(elist) -> SList(List.map (fun s -> to_sexpr symbol s) elist)
    | Ast.Chord(elist) -> SChord(List.map (fun s -> to_sexpr symbol s) elist)
    | Ast.System(elist) -> SSystem(List.map (fun s -> to_sexpr symbol s) elist)
    | Ast.Call(e1, e2) -> SCall(e1, (List.map (fun s -> to_sarg symbol s)  e2))
    | Ast.Let(decs, e) -> let sym = {parent=Some(symbol); identifiers=[]} in
                           let nested_prog = List.fold_left walk_decl {decls=[]; symtab=sym} decs
                           in SLet(nested_prog, to_sexpr sym e)
    | Ast.Print(e)      -> SPrint(to_sexpr symbol e)

and to_sarg symbol = function
    | Ast.Arglit(i)           -> SArglit(i)
    | Ast.Argbool(b)          -> SArgbool(b)
    | Ast.Argvar(s)           -> SArgvar(s)
    | Ast.Argbeat(e, i)       -> SArgbeat(to_sexpr symbol e, i)
    | Ast.Argnote(e1, e2, e3) -> SArgnote(to_sexpr symbol e1, to_sexpr symbol e2, to_sexpr symbol e3)
    | Ast.Argchord(elist)     -> SArgchord(List.map (fun s -> to_sexpr symbol s) elist)
    | Ast.Argsystem(elist)    -> SArgsystem(List.map (fun s -> to_sexpr symbol s) elist)
    | Ast.Arglist(elist)      -> SArglist(List.map (fun s -> to_sexpr symbol s) elist)
    | Ast.Argparens(p)        -> SArgparens(to_sexpr symbol p)

(* Second pass -> use symbol table to resolve all semantic checks *)
and walk_decl_second program = function
    | SVardef(s_id, s_expr) as oldvar ->
        let new_sexpr = (match s_expr with
          SLet(prog, exp) -> SLet(List.fold_left walk_decl_second prog prog.decls, exp)
          | x -> x) in
        let texpr = [get_type false program.symtab new_sexpr] in
        if (s_id.v_type = [Unknown]) then
            let new_type = if (exists_typesig s_id.name program.symtab.identifiers) then
                               let set_type = get_typesig s_id.name program.symtab.identifiers in
                               if diff_types set_type texpr then
                                   (match (List.hd set_type) with
                                    Poly(_) -> texpr
                                    | _ -> raise (Type_mismatch s_id.name))
                               else set_type
                           else texpr in
            let newvar = SVardef({name = s_id.name; pats = []; v_type = new_type;
                                  v_expr = s_id.v_expr}, new_sexpr) in
            replace_vardef program newvar oldvar
        else if diff_types s_id.v_type texpr then
            raise (Type_mismatch s_id.name)
        else program
    | SFuncdec(info) as oldfunc ->
        let types = get_types_p info.s_fname program.symtab in
        let argl = List.length info.s_args in
        let tyl = List.length types in
        let info = {s_fname = info.s_fname; type_sig = info.type_sig; s_args = info.s_args;
                    scope = info.scope; s_value = (match info.s_value with
                        SLet(prog, exp) -> SLet(List.fold_left walk_decl_second prog prog.decls, exp)
                        | x -> x)} in
        if (argl <> tyl - 1) then raise (Pattern_num_mismatch( argl, tyl - 1))
        else let search_decls = List.filter (fun v -> v != oldfunc) program.decls in
            if (List.length search_decls < (List.length program.decls) - 1)
                || (same_pats info search_decls)
            then raise (Multiple_identical_pattern_lists
                        (String.concat " " (List.map string_of_patterns info.s_args)))
            else
                let symtab = (check_pat_types types info)  in
                let newscope = check_ret_type symtab types info in
                let newfunc = SFuncdec({s_fname = info.s_fname; type_sig = types;
                                        s_args = info.s_args; s_value = info.s_value;
                                        scope = newscope;}) in
             replace_funcdec program newfunc oldfunc
  | SMain(expr) ->
      let e_type = get_type false program.symtab expr in
        let new_main = {name = "main"; pats = []; v_type = [e_type]; v_expr = Some(expr)} in
        let program = replace_main program new_main in
          program
      (*  if main_type_check e_type then program else
            raise (Main_type_mismatch (string_of_sexpr expr))
      *)

let has_main program =
  if(is_declared "main" program.symtab) then program
  else raise Main_missing

(* Right now gets called by smurf *)
let first_pass list_decs =
    let program = List.fold_left walk_decl {decls=[]; symtab = global_env} list_decs
    in  program

let second_pass list_decs =
    let program = first_pass list_decs in
        let real_program = List.fold_left walk_decl_second (has_main program) program.decls in
    (print_string "PASSED SEMANTIC CHECKS\n"); real_program.symtab
```

../../Code/semanalyze.ml

```
open Sast
open Util

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
    Semanalyze.second_pass program
```

../../Code/semantic_test.ml

```
(* File: toplevel.ml
 * the toplevel execuatable for SMURF
 *)

open Util
open Interpreter
open Output
open Values
open Lexing


exception Fatal_error of string
let fatal_error msg = raise (Fatal_error msg)

exception Shell_error of string
let shell_error msg = raise (Shell_error msg)

let exec_file config =
    let read_file filename =
        let lines = ref [] in
        let chan = open_in filename in
        (try
            while true; do
                lines := input_char chan :: !lines
            done; []
        with End_of_file ->
            close_in chan;
            List.rev !lines) in
    let fh = read_file config.smurf_name in
    let stdlib = read_file config.std_lib_path in
    let linkedprog = string_of_charlist (stdlib @ fh) in
    let lexbuf = Lexing.from_string linkedprog in
    let pos = lexbuf.lex_curr_p in
    lexbuf.lex_curr_p <- {pos with pos_fname = config.smurf_name};
    try
    let program = Parser.program Scanner.token lexbuf in
    let symtab = Semanalyze.second_pass program in
       (exec_main symtab config)
    with
        Parsing.Parse_error -> fatal_error ("Syntax Error: " ^ string_of_position lexbuf.lex_curr_p)

let _ =
    let interactive = ref false in
    let config = { smurf_name = "smurf.sm";
                   bytecode_name = "a.csv";
                   midi_name = "a.midi";
                   lib_path = "./Lib/CSV2MIDI.jar";
                   std_lib_path = "./Standard_Lib/List.sm"
    } in
    Arg.parse
        [("-i", Arg.Set interactive, "Interactive model");
         ("-o", Arg.String (fun f -> config.midi_name <- f), "Specify output MIDI name");
         ("-l", Arg.String (fun f -> config.lib_path <- f), "Specify the path to the library converting
     bytecode to MIDIs")]
        (fun f -> config.smurf_name <- f)
        "Usage: toplevel [options] [file]";
match !interactive with
      false -> exec_file config
    | true -> ()
```

../../Code/toplevel.ml

```
(* File: util.ml
 * defines some useful stuffs that might be used by other modules
 *)

open Printf
open Lexing

(* If you doing want to see the annoy debug information,
 * simply set debug to be false, the world will be peace
 *)
let debug = false

let trace s = function
    a -> if debug then
            ignore(printf "*** %s\n" s)
         else (); (a)


(* Errors can be handled and will cause the program to terminate *)
exception Fatal_error of string
let fatal_error msg = raise (Fatal_error msg)


type configruation = {
    mutable smurf_name : string;
    mutable bytecode_name : string;
    mutable midi_name : string;
    mutable lib_path : string;
    mutable std_lib_path : string;
}

let rec string_of_charlist = function
    | [] -> " "
    | lst -> String.make 1 (List.hd lst) ^ (string_of_charlist (List.tl lst))

let string_of_position {pos_fname=fn; pos_lnum=ln; pos_bol=bol; pos_cnum=cn} =
  let c = cn - bol in
    if fn = "" then
      "Character " ^ string_of_int c
    else
      "File \"" ^ fn ^ "\", line " ^ string_of_int ln ^ ", character " ^ string_of_int c
```

../../Code/util.ml

```
(* File: values.ml
 * defines the intermediate values smurf evaluates to *)

open Ast
open Sast
open Util
open Printf


exception Interp_error of string
let interp_error msg = raise (Interp_error msg)

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

(* The value of returned by each expression *)
type value =
    | VInt of int
    | VBool of bool
    | VBeat of int
    | VNote of value * value * value
    | VList of value list
    | VChord of value list
    | VSystem of value list
    (*| VFun of pattern list  pattern *)
    | VUnknown

and nm_entry = {
    nm_expr : s_expr option;
    nm_value : value;
}

type enviroment = {
    parent : enviroment option;
    mutable ids : nm_entry NameMap.t;
}

let rec string_of_value = function
    | VInt(x) -> string_of_int x
    | VBool(x) -> string_of_bool x
    | VBeat(x) -> string_of_int x
        (*
        string_of_value i1 ^
        let rec repeat n s =
            if n>0 then
                repeat (n-1) ("." ^ s)
            else s in repeat i2 ""
        *)
    | VNote(pc, reg, beat) -> "(" ^ string_of_value pc
        ^ ", " ^ string_of_value reg ^ ")$"
        ^ (string_of_value beat)
    | VList(vl) -> "[" ^ (String.concat "," (List.map string_of_value vl)) ^ "]"
    | VChord(vl) -> "[" ^ (String.concat "," (List.map string_of_value vl)) ^ "]"
    | VSystem(vl) -> "[" ^ (String.concat "," (List.map string_of_value vl)) ^ "]"
    (*
    | VFun(name,fsig,fdecl) ->
        (match fsig with
              Tysig(name,types) -> (name ^ " :: "
                 ^ String.concat " -> " (List.map Ast.string_of_types types) ^ "\n\t     ")
            | _ -> interp_error ("Unexpected type for Tsig"))
        ^ (String.concat "\t     " (List.map Ast.string_of_fdec fdecl))
    *)
    | _ -> "Unresolved"

(* show the environment to std out *)
let rec show_env env = match debug with
      true ->
        (match env.parent with
              None -> printf "GlobalE: \n"; NameMap.iter
              (fun key {nm_value=v} -> print_string ("\t" ^ key ^ " -> "
              ^ string_of_value v ^ "\n")) env.ids
            | Some x -> printf "LocalE: \n"; NameMap.iter
              (fun key {nm_value=v} -> print_string ("\t" ^ key ^ " -> "
              ^ string_of_value v ^ "\n")) env.ids; show_env x)
    | false -> ()

let rec string_of_env env = (match env.parent with
      None -> "GlobalE: \n"
        ^ (NameMap.fold (fun key {nm_value=v} str -> str ^ ("\t" ^ key ^ " -> "
          ^ string_of_value v ^ "\n")) env.ids "")
    | Some par -> "LocalE: \n"
        ^ (NameMap.fold (fun key {nm_value=v} str -> str ^ ("\t" ^ key ^ " -> "
          ^ string_of_value v ^ "\n")) env.ids "") ^ string_of_env par)
```

../../Code/values.ml

```
SOURCES = scanner.mll \
          parser.mly \
          sast.ml \
          ast.ml \
          semanalyze.ml \
          parser.ml \
          scanner.ml \
          parser_test.ml \
          semantic_test.ml \
          interpreter.ml \
          util.ml \
          toplevel.ml \
          values.ml \
          output.ml

OCAMLBUILD = ocamlbuild

all:
  $(OCAMLBUILD) parser_test.native semantic_test.native toplevel.native

clean:
  $(OCAMLBUILD) -clean
  rm -f a.csv a.midi
```

../../Code/makefile

```
# For those machine doesn't have ocamlbuild, build the project with this makefile

OBJ=ast.cmo \
  sast.cmo \
  semanalyze.cmo \
  scanner.cmo \
  parser.cmo \
  util.cmo \
  parser_test.cmo \
  semantic_test.cmo \
    interpreter.cmo \
    toplevel.cmo \
    values.cmo \
    output.cmo \
    printer.cmo

SMURF=semantic_test

FLAGS:=-g

$(SMURF): $(OBJ)
  ocamlc -g -o parser_test util.cmo parser.cmo scanner.cmo ast.cmo parser_test.cmo
  ocamlc -g -o semantic_test util.cmo parser.cmo scanner.cmo ast.cmo sast.cmo semanalyze.cmo
     semantic_test.cmo
  ocamlc -g -o toplevel util.cmo parser.cmo scanner.cmo ast.cmo sast.cmo semanalyze.cmo values.cmo output
     .cmo interpreter.cmo toplevel.cmo

printer: $(OBJ)
  ocamlc -o printer.cma -a util.cmo ast.cmo sast.cmo values.cmo printer.cmo

.SUFFIXES: .ml .cmo .cmi .mll .mly .mli
.PRECIOUS: %.ml %.mli %.cmo

.ml.cmo:
  ocamlc -c $(FLAGS) $<

.mli.cmi:
  ocamlc -c $(FLAGS) $<

.mll.ml:
  ocamllex $<

.mly.ml:
  ocamlyacc -v $<

.mly.mli:
  ocamlyacc -v $<

clean:
  rm -f *.cmi *.cmo parser.ml scanner.ml *.output parser.mli parser_test semantic_test toplevel *.cma


parser_test: $(SMURF)
  ./parser_testall.sh

# Generated by ocamldep
ast.cmo:
ast.cmx:
interpreter.cmo: values.cmo util.cmo sast.cmo output.cmo ast.cmo
interpreter.cmx: values.cmx util.cmx sast.cmx output.cmx ast.cmx
output.cmo: values.cmo util.cmo ast.cmo
output.cmx: values.cmx util.cmx ast.cmx
parser.cmo: util.cmo ast.cmo parser.cmi
parser.cmx: util.cmx ast.cmx parser.cmi
parser.cmi: ast.cmo
sast.cmo: util.cmo ast.cmo
sast.cmx: util.cmx ast.cmx
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
semanalyze.cmo: util.cmo sast.cmo ast.cmo
semanalyze.cmx: util.cmx sast.cmx ast.cmx
semantic_test.cmo: util.cmo semanalyze.cmo scanner.cmo sast.cmo parser.cmi
semantic_test.cmx: util.cmx semanalyze.cmx scanner.cmx sast.cmx parser.cmx
parser_test.cmo: scanner.cmo parser.cmi ast.cmo
parser_test.cmx: scanner.cmx parser.cmx ast.cmx
toplevel.cmo: values.cmo util.cmo semanalyze.cmo scanner.cmo parser.cmi \
            output.cmo interpreter.cmo
toplevel.cmx: values.cmx util.cmx semanalyze.cmx scanner.cmx parser.cmx \
            output.cmx interpreter.cmx
util.cmo:
```

```
util.cmx:
values.cmo: util.cmo sast.cmo ast.cmo
values.cmx: util.cmx sast.cmx ast.cmx
printer.cmo: util.cmo ast.cmo sast.cmo values.cmo
printer.cmx: util.cmx ast.cmx sast.cmx values.cmx
```

../../Code/build.mk

References

[1] A. Appleby, “Accidentals.” `http://www.music-mind.com/Music/mpage4.HTM`, Sept. 2013.

[2] M. DeVoto, “Twelve-tone technique: A primer.” `http://www.tufts.edu/~mdevoto/12TonePrimer.pdf`, Sept. 2013.

[3] L. J. Solomon, “Symmetry as a compositional determinant.” `solomonsmusic.net/diss7.htm#Webern`, 1997.

[4] S. Steffes, “Csv2midi.” `https://code.google.com/p/midilc/source/browse/trunk/CSV2MIDI/CSV2MIDI.java`, June 2003.