

# 15-212 Fall 2006

## Assignment 5: MLMusic Compiler

Due: 11/19/06  
Maximum points: 100 + 15 EC

### Important Read Me First!

- Start early. This assignment is hard.
- You will make changes to multiple files and create some of your own in addition to the templates. Copy these files into your handin directory at: `/afs/andrew/course/15/212/studentdir/<your_andrew_id>/assn5`
- **Place the answers to written problems in the proof.sml file.**
- Your functions must be contained in our structures and use the types defined in the corresponding signatures. Do not edit the structures or signatures. There is no check script. Ill-typed code will not compile in a structure context. Non-compiling code will not be graded. Simply entering **CM.make “sources.cm”** into the SML interpreter in the project directory will compile all of the code. This must return **val it = true : bool** or else your code does not compile.
- You can use any library functions. If you use a library function, use the dotted notation to access functions inside a structure scope. **Do not open library structures into your structures or into the top level. Do not shadow library types like option and order.**
- Additionally you may use any imperative features you want except while, callcc (SMLNJ.cont), CML, Posix, Lex or Yacc.
- Comments and specifications for all functions will be graded.
- See the course syllabus for policies on cheating and late homework.
- The TAs in charge of this homework are Uland Wong (uyw@andrew.cmu.edu) and Deepak Garg (dg+@cs.cmu.edu). Don't spam us, Blackboard is your friend.

## 1. Introduction

For this project we will be designing a compiler for MLMusic, a simple tagged language used to express sheet music. We will experience the majority of the compilation process from testbed writing to viewing “assembly” output, in which the end product will be a midi file (sort of like an mp3) representing an input song. This homework is structured differently from previous homeworks. Each consecutive problem builds on the last, and there are a lot of conceptual questions to make sure you are thinking before and while coding. Additionally, the nature of compilation makes this homework largely open-ended. You have a lot of freedom to explore implementation issues and also shoot yourself in the foot. To get started:

1) Get the `assn5handout.tar.gz` file from Blackboard

2) Untar it inside some working directory:

```
%unzip assn5handout.zip -d ~/your_directory
```

3) To compile your source files, start SML and run `CM.make “sources.cm”`;

```
Standard ML of New Jersey, v110.59 [built: ...]  
- CM.make “sources.cm”;
```

You now have the required files for assignment 5. But, don’t start coding yet! Read ahead.

## 2. Musical Notation

Due to space constraints in this document, we have included a lengthy primer to musical notation as a separate handout in **music\_primer.pdf**. This is mainly for your own edification so that you may realize how the language ties in to actual music. Reading it and/or becoming an expert in music is not necessary to complete the assignment but you are encouraged to read the document as it may quickly clear up syntactic confusions.

## 3. The MLMusic Language

The MLMusic language is a simple tag-based language to express musical notes and the accompanying information needed to generate meaningful musical sound. It is based on traditional music engraving notation, in a form compatible with keyboard input. The complete syntax of the MLMusic is given below in Backus-Naur Form (BNF). Any valid code specified with the MLMusic language will be a subset of these rules. If you do not remember BNF from earlier in the semester, please brush up on it before reading further.

The top level of the language in BNF form is given by the `MLMUSIC` construct which is distinct from the structure of the same name in the coding portion of the assignment. The correspondence to natural music notation is obvious. There are some artificialities, however, that were introduced into the language to simplify writing the compiler. For example, we only allow parallelism for single notes (or repeats) and not arbitrary sequences of notes. Sequential parallelism is a feature that exists inherently with multiple staves, and forcing this constraint erases an ugly recursive loop in our syntax. Additionally, the grammar is written to force all lists to be of length 1 or greater, which

will simplify the type checker problem later on. The meanings of specific computer symbols like “#” are given in the section when you write the lexer.

### BNF Specification for MLMusic

```
MLMUSIC ::=
    TEMPO=NUM TS=(NUM, NUM) KS=(PITCH, MODIFIER) STAFF_LIST

STAFF_LIST ::= STAFF | STAFF STAFF_LIST

STAFF ::= HAND CLEF SEQUENCE

HAND ::= L NUM | R NUM

MODIFIER ::= # | * | _

SEQUENCE ::= SEQ [ MEASURE_LIST ]

MEASURE_LIST ::= MEASURE | MEASURE MEASURE_LIST

MEASURE ::= NOTE MEASURE | PARALLEL MEASURE | .

PARALLEL ::= {NOTE_LIST}

NOTE_LIST ::= = NOTE | NOTE NOTE_LIST

NOTE ::= <PITCH, MODIFIER, DURATION, OCTAVE>
        | NUM<PITCH, MODIFIER, DURATION, OCTAVE>

CLEF ::= TRB | BAS

PITCH ::= C | D | E | F | G | A | B | _

DURATION ::= NUM | NUM .

OCTAVE ::= NUM
```

Note that the grammar of natural numbers (NUM) is omitted for clarity. If you notice, some of the constructs are not truly necessary. For example, we can collapse SEQUENCE which includes only a single non-terminal (MEASURE\_LIST) and use only the MEASURE\_LIST construct instead. However, this method makes the grammar clearer from an abstract syntax (defined in section 7) perspective and allows clean introduction of terminals.

The core of the language is basically the note, which has several attributes that dictate how a note is played (tonality, duration, etc). The other constructs in the language define groupings of notes, either sequentially or in parallel. Wrapping these multi-note constructs are measures which dictate the overall display and grouping of the note constructs on paper, but have little actual effect on the flow or sound of the music. Above this, a staff includes many measures and is essentially an independent piece of music, many of which can be played at once (think two hands or multiple people on the piano). Finally, the top level of the language groups some common definitions that are constant throughout the piece, such as tempo and key and time signature. These attributes are similar to those presented in the music primer with some simplifications. The primer specifies tempo as a number of beats per minute relative to a note of certain duration. We do not allow specification of such a note. Our tempo is always relative to the quarter note.

We also only allow a single time signature for the whole piece as opposed to each individual staff. This greatly simplifies type checking. The key signature is also global for an entire piece of music and only allows a single key and modifier. Its effect is primarily for type setting and does not have an effect on the sound of the piece.

Below is example syntax for the first few notes of *Mary Had a Little Lamb* written using the MLMusic language:

```

TEMPO = 120
TS = (3, 4)
KS = (A, _)
  L1 TRB SEQ
  [
    <A, #, 4, 4><G, #, 4, 4><F, #, 4, 4>.
    <G, #, 4, 4><A, #, 4, 4><A, #, 4, 4>.
    <A, #, 4, 4><_, #, 4, 4><G, #, 4, 4>.
    <G, #, 4, 4><G, #, 4, 4><_, #, 4, 4>.
    <A, #, 4, 4><A, #, 4, 4><A, #, 4, 4>.
  ]

```

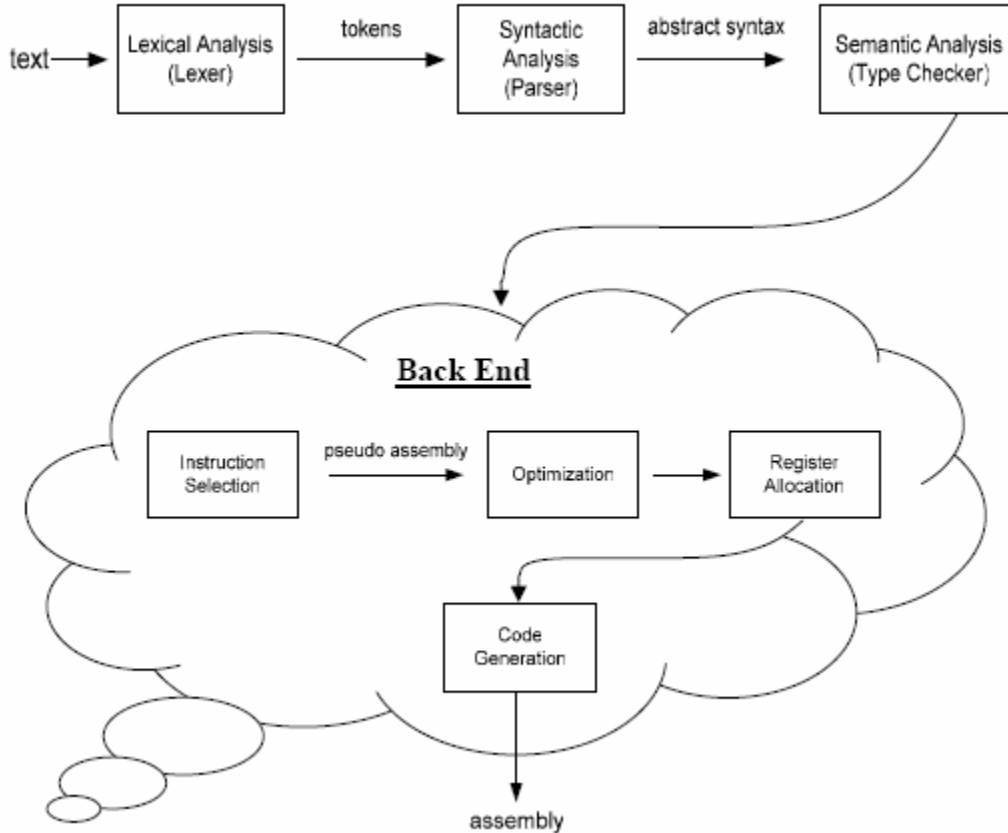
**Problem 3.1 (5 points)**

What is the regular expression of the language accepted by the NOTE construct?

**4. The MLMusic Compiler**

The MLMusic compiler operates on “code” written in the MLMusic language. The MLMusic compiler is a full strength compiler as far as functional units are concerned. Like a C or SML compiler, it must map from some series of input characters to a specific bit output that will control a machine in the intended manner. It must also deal with faulty input, non-bijective mapping from one language to another, and a wide range of other issues. Unlike a compiler for a more complex language; however, the MLMusic compiler does not have to deal with hardware implementation, error recovery, control flow or memory management. Below, is a rough “road map” of the flow of data in a compiler. Your work problems in the assignment will closely mirror the logical components in the front end.

## Front End



Compilers are usually split into two disjoint sections which can often be considered programs themselves: the front end and the back end. Normally, there is a single static front end and multiple versions of the back end that depend on the *target* machine you are compiling. As such, our design will primarily concern itself with the front end.

The front end of a compiler transforms ASCII text input (usually in the form of files) into some intermediate representation where it is manipulated by software. Usually, there are several intermediate representations corresponding to various stages in the front end leading to a final representation called the abstract syntax tree. The various representations of the code in the MLMusic front end are char lists (lexical analysis), token lists (syntactic analysis), and abstract syntax (type checker).

The back end of a compiler converts abstract syntax that has been checked for errors into a form that the operating environment can execute. For a C compiler on modern personal computers, this form may be x86 (an assembly language) which must be further “assembled” into raw bits; a java compiler may translate java code into the form of bytecodes for a virtual machine; and for particularly old machines, this form may be the patterns on a punch card. We will be translating our MLMusic code into another language: lilypond. This will serve as our “assembly language” while the GNU lilypond environment (which actually does the job of writing the midi file) will be our assembler and our virtual machine will of course be a capable media player.

## 5. The Compilation Environment

The framework for the MLMusic compiler is a series of structures. There is a single structure per file, which is named the same as the file except in all caps. The top level structure is the structure `TOP` located in `top.sml`. Running `CM.make "sources.cm"` with `sml` in your project directory will load `TOP` into the top level environment. From there, you will have access to several important functions which will help you in the debug process for the various parts of the compiler. Below is the signature for `TOP`:

### Important Functions in TOP

<pre>val install_lilypond: unit -&gt; unit</pre>	Installs the lilypond software on your computer.
<pre>val loadfile : string -&gt; string</pre>	Reads in a file containing your MLMusic code for compilation. The argument is the path to the file and a string representation of the file is returned.
<pre>val writefile : (string * string) -&gt; unit</pre>	Writefile writes a string given by the first argument to a file given by the second argument. A “.ly” extension is appended to the file name.
<pre>val run : string -&gt; string</pre>	Run runs the lexer, parser, type checker, and code generation on an input string which is MLMusic code read in using loadfile.
<pre>val compile : (string * string * bool) -&gt; unit</pre>	Compile automates the loadfile, run and writefile functions. The first argument is a string path to the input file, the second argument is the string path to the output file (will have “.ly” appended to path) and the Boolean specifies whether the PDF file will be displayed after compilation.

Note that there are also other functions in the `TOP` structure. These functions were written by the author when he designed the compiler. They proved very useful in debugging the various stages. The specifications of these functions are purposefully not given. You do not have to use them in writing your code and you do not need to make an effort to understand what they do. Since the author is primarily a windows user (yes we exist), automatic installation of the lilypond environment and code compatibility is also provided for those wishing to code in windows (we will not provide tech support).

The other structures in the project are `LEX` given in `lex.sml`, `PARSE` given in `parse.sml`, `TC` given in `tc.sml` (type checker), `CODEGEN` given in `codegen.sml` and finally `MLMUSIC` given in `mlmusic.sml`. `LEX`, `PARSE`, and `TC` are stubs meant to be filled in by the student. `CODEGEN` is the “back end” of the compiler and you should not be the least bit concerned with it (at least in what is graded for this assignment). Finally, `MLMUSIC` is the file you should become intimately familiar with during the course of this project. The

structure defines all of the datatypes you will be using between various stages of the assignment. Comments in the datatypes (yes, they were put in by me because I comment) will be very helpful in telling you where and how certain pieces fit together. Additionally, there are a wide variety of helper functions that were written by the author for his implementation of the compiler. Once again, you do not need to use any of these functions (although for the MLMusic library functions, this is highly recommended) and you will most likely not use all of them. These functions all have a usage model in comments. Pay close attention to the `MLMUSIC` structure.

There is one last thing to note about coding with structures in `sml`. You never know what functions are bound at the top level and you do not know which structures have been opened at the top level (possibly shadowing your desired function), so please do not assume anything. If you see at the top of the files we have included some declarations like:

```
structure M = MLMUSIC
structure L = LEX
structure P = PARSE
structure T = TC
structure C = CODEGEN
```

Please use the dot notation (`C.compile`, `M.BAR` etc) to refer to functions and datatypes contained in other structures. Do not just type `compile` or `BAR` and assume it is bound and do not clutter the top level by writing `open M`, etc.

## 6. Lexing and Tokens

The first phase of any compiler or interpreter system is the lexer (lexical analyzer). Lexing is the process of turning a stream of ASCII characters into tokens (sometimes called lexemes) which are a class of representable symbols in the language. The ASCII stream can either be the contents of a code file or user input from a command prompt. This phase also selects language-specified keywords from the code and separates these from other allowable elements. Lexing is useful because it reduces the problem space and separates much of the mundane string manipulation from the parser, where it is much easier to make errors. There are many professional lexer-generating programs available which will automatically spit out code for a lexer given a short specification. The most popular is Lex for the C language, of which there is a port for SML, called ML-Lex.

The MLMusic compiler will read in a file as a c-style string. This string format preserves all formatting including newlines, tabbing, spaces and slashes as unique characters (consult an ASCII chart). We can read in a file using the given function `loadfile`:

```
val mycode = TOP.loadfile "mymusicfile.mlm";
```

Calling `String.explode` on the output of `loadfile` will convert the string into a char list. The lexer will operate on the char list and convert it to a token list given by the `MLMUSIC.TOKEN` type. The `TOKEN` datatype is shown below (taken from the `MLMUSIC` structure):

```

datatype TOKEN =
  LANGLE (*<*) | RANGLE (*>*) | LBRACE (*[*] | RBRACE (*]*) |
  LCURLY (*{*) | RCURLY (*}*) | LPAREN (* ( *) | RPAREN (* ) *) |
  NOTE of char (*A,B,C etc*) | HAND of (char * int) (*L, R*) |
  UNDERSCORE (*_*) | HASH (*#*) | SPLAT (* * *) | EQUALS (*=*) |
  NUM of int (*1,13,54 etc*) | DOT (* . *) | BAR (*|*) |
  SEQ (*SEQ*) | TS (*TS*) | TEMPO (*TEMPO*) | KS (*KS*) |
  TRB (*TRB*) | BAS (*BAS*)

```

In general, the lexer is a one to one mapping. For each character input, it will return a single corresponding token. For example, if the character #"<" is detected on the input, it should output the `MLMUSIC.LANGLE` token. The notable exceptions are numbers and keywords. In tokenizing numbers, we will want to form a token consisting of several ascii numeric digits. For example, the lexer should output `[#"1", #"2", #"0"]` as a single token `MLMUSIC.NUM 120`, and not `MLMUSIC.NUM 1`, `MLMUSIC.NUM 2` and `MLMUSIC.NUM 0` separately. Keywords follow a similar rule.

One of the concerns with lexing is what to do with white space, tabs and new lines. These "delimiters" are handled differently by each language. Visual Basic treats new lines as end-of-statements while C and Java allow any amount of white space so long as it does not break up an identifier or a keyword (special words reserved by the language). MLMusic will follow the C white space rule. We will allow any amount of white space as long as it does not break up a keyword (we do not have identifiers). Additionally, we will regard the comma character as a delimiter.

### Problem 6.1 (15 points)

Write a lexer for MLMusic files in the LEX structure in `lex.sml`:

```

val lex : char list -> MLMUSIC.TOKEN list
exception InvalidChar of char

```

Your job is to convert characters to the appropriate tokens, handle delimiters, and correctly tokenize numbers and keywords. The lexer should ignore syntax completely, only checking if a character is valid for the language. If an invalid character is detected, raise the `InvalidChar` exception with the argument being the offending character. The `MLMUSIC` structure contains functions that you may find useful in building your lexer (I know I did). It also contains in comments, the full mapping between characters and tokens (also given above). You must use our character mapping, do not define your own. You cannot use ML-LEX in your implementation.

### Examples:

The string `"SEQ <10 A>"` should be tokenized into the following list:

```
[M.SEQ, M.LANGLE, M.NUM 10, M.NOTE #"A", M.RANGLE]
```

The string `"10 A B [ %"` should raise `InvalidChar #"%"` because the percent symbol is not valid in our language.

**Problem 6.2 (5 points)**

We were very tricky and specified the language such that the underscore character `#"_"` represented both a rest and a natural key signature. If you wrote the lexer correctly, it would have been impossible to ever have the following token construct: `M.NOTE("#"_"`). Why?

**7. Parsing and Abstract Syntax Trees**

According to Wikipedia, "parsing is the process of analyzing a continuous stream of input (read from a file or a keyboard, for example) in order to determine its grammatical structure with respect to a given formal grammar." In short, the parser takes the token stream and changes it to a mathematically useful form, on which calculations can then be performed. This includes stripping out unnecessary symbols such as braces and semicolons, which are useful mainly for human readability. The useful information is then grouped into expressions, statements, operations, declarations and other logical divisions based on the forms specified in the grammar. The parser is so important to language systems (and such a large part of compiler error) that the commercial application YACC (yet another compiler-compiler) was created to allow automatic, error-free generation of parsers.

Because the language may include expressions inside of expressions, one can easily see that it is best represented with a recursive data structure. Indeed, the output of the parser is a tree (or graph) of labeled nodes with variable children, called an Abstract Syntax Tree or AST. In this document we will use AST to denote abstract syntax which is expressly different from the form of code in a language (concrete syntax) which we have thus far called just syntax.

The AST for the MLMusic language is, unsurprisingly, much less complex than the AST for SML. However, it is still quite daunting as it is probably your first brush with multiple, mutually-dependant, recursive data structures. As with any BNF form definition, there are also terminal symbols and expressions in the MLMusic AST. Study the MLMUSIC structure and be sure you understand what all of the datatypes beginning with MLMUSIC.AST and ending with MLMUSIC.STAFF are for. It should be clear that the top level is the MLMUSIC.AST type which has a single constructor called `Music`. The sml datatype for abstract syntax is given:

```
datatype AST =
    Music of (int * (int * Duration) *
             (Pitch * Modifier) * Staff list)

and Hand = LEFT of int | RIGHT of int

and Pitch = C | D | E | F | G | A | B | REST

and Clef = TREBLE | BASS

and Modifier = SHARP | FLAT | NATURAL

and Duration = Fraction of (int * bool)

and Octave = OCT of int
```

```

and Note = Tone of (Pitch * Modifier * Duration * Octave)
           | Repeat of (int * Pitch * Modifier * Duration * Octave)

and Parallel = Simul of (Note list)

and Measure = Bar_seq of (Note * Measure)
              | Bar_par of (Parallel * Measure)
              | BAR_END

and Sequence = Seq of Measure list
and Staff = Collection of (Hand * Clef * Sequence)

```

The goal of the MLMusic parser will be to map a list of tokens from the lexer to a recursive data structure defined by the AST. The parser will therefore be a series of mutually recursive functions, of which each maps a bit of the token list to a particular AST expression and calls its sister functions recursively to fill in missing sub-expressions. The sub-expressions get smaller and smaller during each recursive call until the functions that map terminal AST symbols correspond to single tokens directly. For example you might write a function called `parse_note` that is responsible for extracting a note expression from a token list. This function might call `parse_pitch`, `parse_modifier`, `parse_duration` and `parse_octave` in order, each which would consume more of the token list and return the resulting expressions. Once all of the sub expressions are filled in, we can return `M.Tone(p,m,d,oc)` as well as what is left of the token list from parsing the note.

Mapping to the AST should reveal syntax errors, especially if the mapping from Tokens to AST is unique (and it is for our language). Often, compilers will automatically fix syntax errors, but that is outside the scope of this course. If you encounter a syntax error, an order of tokens that does not make any sense, raise the `Syntax` exception with the argument being the token list that remains to be parsed.

### Problem 7.1 (25 points)

Write a parser for MLMusic files in the PARSE structure in `parse.sml`:

```

val parse : MLMUSIC.TOKEN list -> MLMUSIC.AST
exception Syntax of MLMUSIC.TOKEN list

```

Your job is to convert the token list output from the lexer in the previous section, to an abstract syntax tree. Raise the `Syntax` exception with the remainder of the token list if a syntax error is detected.

The parser is very tricky to code. Pay close attention to comments and other hidden goodies in the compiler environment and make sure you understand ASTs before you start coding. Requiring the use of the `Syntax` exception with an embedded token list has a two fold benefit. Firstly, it is very helpful for debugging your code (which you must do!). In fact, we have kindly included functions in the `TOP` structure to help you with debugging using the exception. Secondly, if you can remember exceptions and continuations you will realize that exceptions can be use to control the flow of a program to try one thing and then another. The type of the `Syntax` exception tells us it would be ideal to use as a “failure exception.” Although we recommend this style of programming for the assignment, you are not required to use exceptions in this way. An alternative would be to use continuations. You cannot use ML-YACC in your implementation.

### Examples:

The following list of tokens output by the lexer:

```
[M.LANGLE, M.NOTE #"A", M.HASH, M.NUM 4, M.NUM 4, M.RANGLE]
```

should be parsed as a single note:

```
M.Tone (M.A, M.SHARP, M.Fraction(4,false), M.OCT 4)
```

The following list of tokens output by the lexer:

```
[M.TEMPO, M.RANGLE, M.NUM 120]
```

should raise the `Syntax` exception with the list `[M.RANGLE, M.NUM 120]` since only an equals sign can logically follow the `M.TEMPO` token.

#### Problem 7.2 (5 points)

Notice the following construct in the AST:

```
Repeat of (int * Pitch * Modifier * Duration * Octave)
```

It corresponds to the concrete syntax given by:

```
NUM<PITCH, MODIFIER, DURATION, OCTAVE>
```

It means to play the same note multiple times. This shortening of code by including redundant constructs is called syntactic sugar. Instead of specifying the repeat type, what is another way we could have implemented this functionality? Hint: this is not a trick question. The answer is very simple.

#### Problem 7.3 (5 points)

For practical reasons, why do we specify the `.` (dot) character and corresponding `BAR_END` terminal as delimiters when constructing a measure? Think about the regular expressions of sequences in the current language:

```
seq = ((<note> + {parallel})*.)*
```

Hint: staring at the above regular expression is really all you need to figure this out.

#### Problem 7.4 (5 points)

If you notice the following AST construct:

```
MEASURE ::= NOTE MEASURE | PARALLEL MEASURE | .
```

This implies that parsing measures is left associative. We could very easily have defined them to be right associative with the construct:

```
MEASURE ::= MEASURE NOTE | MEASURE PARALLEL | .
```

What effect, if any would this have on your parser?

## 8. Typechecking and Semantic Analysis

From Wikipedia, “A type system defines how a programming language classifies values and expressions into types, how it can manipulate those types and how they interact. A type indicates a set of values that have the same sort of generic meaning or intended purpose.”

Not every expression allowed by the syntax of `MLMusic` makes sense. For example, if we play two notes in parallel, we must be sure they are of equal length, or

else you will get some very weird conditions (like when do we start playing the shorter note relative to the longer one?). Thus, we need some facilities to help determine which MLMusic expressions are well-typed and hence have a sensible runtime behavior. In short, before we attempt to compile something, we had better be sure it can be compiled and that it will run without any type errors when compiled.

Our type checking problem is actually quite similar to the one posed by sml, however greatly simplified. One can surmise that the only “type” the MLMusic language has is that of durations, which we will represent using the sml `real` type as length in seconds. Sub-expressions of certain duration are built up into larger constructs which have a total duration that is some function of the durations of its parts. Ultimately, the type of an entire MLMusic file will be a single `real`, representing how long that piece plays for. Compare this with sml, in which the final type of an expression is a type within the set of allowable types. If the only allowable type is a `real`, we have compressed the sml typechecking problem into our MLMusic space. Along the way, some expressions will require other bounds checks unrelated to their durations, however. These checks are just as important as checking the durations. The typechecking rules for the MLMusic language are given below:

### MLMusic Type Checking Rules

- A duration value must be a strictly positive power of 2 between 1 and 64 inclusive
- All notes in parallel must have the same duration
- An octave value must be between 1 and 8 inclusive
- All bars have a duration that matches the time signature
- All bars have the same length
- The repeat construct cannot be used inside a parallel construct
- Each staff identification must occur only once
- All staves must have the same total length
- If there is more than 1 staff in the piece, each hand must have at least 1 staff
- The tempo must be valid. Valid tempos are between 40 and 208 (measured in beats per minute) inclusive and must be divisible by 4.
- The key signature does not allow `M.REST * Modifier` as a possible value. An atonal piece is achieved by using the `M.NATURAL` modifier on a non-rest note.
- The duration in the time signature must not be a dotted note (the Boolean must be false)

#### **Problem 8.1 (25 points)**

Write a type checker that implements the rules given above for MLMusic abstract syntax in the TC structure in the `tc.sml` file.

```
val type_check : MLMUSIC.AST -> real
datatype TErr = TMPO_T | KS_T | DUR_T
              | OCT_T | SIMUL_T | BAR_T | STAFF_T
exception TypeError of TErr
```

It is your job to implement the typing rules in the MLMusic framework. If the code passes all the rules, return the total length of the piece. If the code fails any of the rules, raise an `TypeError` exception with the argument being of datatype `TErr`. The terminals in

`TErr` denote what sort of rule failed, whether it was checking durations, bars, or the tempo etc that did not pass. Like the parser, the implementation of the typechecker should be a series of mutually recursive functions that crawls the AST.

Note that in checking the identifiers for staves, we will need to keep track of the hand and the number of the staves that have already been declared. In order to facilitate this, we have written a dictionary implementation for use as a “variable” environment in `dict.sml`. You do not have to use this. You may write your own. However, if you choose to write your own implementation, you must write it inside the `Dict` structure in `dict.sml` (there is no defined signature). Once again, you may find helper functions in the `MLMUSIC` structure helpful.

Lastly, please use the functions in the Real basis library such as `Real.==` for equality, `Real.+` for addition and so on. Do not attempt to use the polymorphic equality operator.

### Examples:

The following note construct should return a real with value 0.25, meaning that it has a duration of one quarter of a second:

```
M.Tone (M.A, M.SHARP, M.Fraction(4,false), M.OCT 4)
```

The following note construct should raise the `TypeError` exception with the `DUR_T` value because the duration is not a valid positive power of 2:

```
M.Tone (M.A, M.SHARP, M.Fraction(13,false), M.OCT 4)
```

### **Problem 8.2 (5 points)**

In many languages (probably not one you are familiar with) there is the concept of a subroutine. The subroutine construct allows a batch of code to be modularized such that it can be run when the name of the subroutine is invoked. Unlike a function, a subroutine does not accept arguments or return values. Let us assume that we want to extend the MLMusic language with subroutines that allows us to play a whole series of notes simply by using an identifier that is declared globally. How does the type checking problem change? Currently, the only type in our language is `real`. Do we need more types now that we have more complex structures? Is a single global environment sufficient or do we need nested environments?

## **9. Code Generation and Submission**

After you have finished writing the front end, you will wish to conduct full system tests to make sure that your integrated code is correct. This can be done by running code generation on the output of the front end and compiling the AST into an adobe PDF file and a midi file, both of which can be examined in their respective viewers. You should never have to edit the code generation structure and we will be grading you using our own default `codegen.sml` file.

The following steps will compile the example `mary.mlm` file:

```
- CM.make "sources.cm";  
[autoloading]  
[library $smlnj/cm/cm.cm is stable]  
.  
.  
.  
[New bindings added.]
```

```

val it = true : bool
- TOP.compile ("mary.mlm", "mary", false);
val it = () : unit

```

This should compile the mary.mlm file into a mary.ly file and if your lilypond environment is setup correctly, it should run lilypond on the .ly file and generate mary.pdf and mary.midi. Keep in mind that there are other helpful testing functions in top, should you need to work out bugs in your code. An example mary.ly file is shown:

```

\score {
<<
  \new Staff {
    \time 3/4
    \clef bass

    ais4 gis4 fis4
    gis4 ais4 ais4
    ais4 r4 gis4
    gis4 gis4 r4
    ais4 ais4 ais4
  }
>>

\midi {\tempo 4=120 }
\layout { }

```

A note (no pun intended) about the engraving style in the PDF file, if there are several of the exact same notes in a row as a part of a measure, only the attributes for the first note are printed, and the attributes for the following notes are assumed. Once the end of a bar is hit, the relative notation is automatically reset and the attributes are printed for the first unique note in the next bar.

### **Problem 9.1 (5 points)**

In the `spec.sml` file document your major design decisions, difficulties, and thoughts regarding your compiler implementation. Give a general overview about how you approached the lexing, parsing and typechecking problems and also specific thoughts on whether what you did for each was optimal or non-optimal.

If you feel you that you have a tested and working compiler implementation and are ready to submit please make sure you have the following files:

```

-lex.sml (lexer)
-parse.sml (parser)
-tc.sml (typechecker)
-dict.sml (if you wrote your own dictionary implementation)
-proof.sml (for written problems)
-spec.sml (your design decisions)

```

Once again, we will check all these files for proper coding style and comments and **documentation is worth some unspecified number of points**. Additionally, if

something does not work, you are advised to comment in length about any sort of thoughts or information you have.

Submit these 6 files into your assn5 handin directory along with any \*.mlm files you wish to get credit for (see section 10 below). Your handin directory is located at: `/afs/andrew/course/15/212/studentdir/<your_andrew_id>/assn5`

## **10. Class Contest**

### **Problem 10.1 (15 points Extra Credit)**

Submit your best compositions as \*.mlm files as a part of your assignment hand-in. We will compile the files and listen to your music. The ones we like will get an arbitrary amount of extra credit up to 15 points and may be played in class.