Clay Bindings Generator

The bindings generator should accept a header file as input, run the preprocessor directives, and in the resulting compilation unit, the following items should be converted.

Function Declarations

C Input
```
RETURNTYPE foo(ARGTYPE1 argname1, ARGTYPE2 argname2);
```

Clay Output
```
external foo(argname1:CONVERTED_ARGTYPE1, argname2:CONVERTED_ARGTYPE2) :
CONVERTED_RETURNTYPE;
```

In C, specifying the argument variable name is optional, whereas in Clay, it is mandatory. In case it is missing in the C declaration, create a dummy name for Clay.

Global Variables Declarations

C Input
```
VARTYPE varname;
```

Clay Output
```
external varname : CONVERTED_VARTYPE;
```

Struct Definitions

C Input
```
struct Node {
    FIELDTYPE1 fieldName1, fieldName2;
    FIELDTYPE2 fieldName3;
}
```

Clay Output
```
record Node {
    fieldName1: CONVERTED_FIELDTYPE1;
    fieldName2: CONVERTED_FIELDTYPE1;
    fieldName3: CONVERTED_FIELDTYPE2;
}
```

Converting Function Pointers

Most types have straight forward mappings in Clay. Function pointers may be a little tricky.

C Input
```
int (*compare_func)(int *a, int *b)
```

Clay Output
```
CCodePointer[Pointer[Int], Pointer[Int], Int]
```

In general, C function pointers have the Clay type: CCodePointer[ARGTYPE1, ARGTYPE2, ..., RETURNTYPE]

Enumerations

C Input
```
enum Something {
    FOO,
    BAR,
    BAZ = 234
}
```

Clay Output
```
static FOO = 0;
static BAR = 1;
static BAZ = 234;
```

Preprocessor Defines

In many C libraries, preprocessor #defines are used to define symbolic names for constants. Without these #defines, the C library becomes difficult to use since the programmer will have to use hard to remember numbers. So, it's important to have these #defines available as Clay constants.

C Input
```
#define MAX_PATH 255
```

Clay Output
```
static MAX_PATH = 255;
```

Typically, after running the preprocessor, #defines are not available. So, it may be difficult to support preprocessor defines. Let me know if there is a way to achieve this.