C Language Reference Manual

I. Introduction - **Jon**
II. Lexical Conventions - **George**
    A. Tokens
    B. Comments
    C. Patterns: intro regex, css
    D. Identifiers
    E. Keywords
    F. Constants
    G. Integer Constants
    H. String Literals
    I. Double Constants
III. Syntax Notation - **Ethan**
    A. Meaning of identifiers
    B. Basic Types
    C. Type Qualifiers
    D. Conversions
    E. Integral Promotion
    F. Integral Conversion
    G. Integer and Double
    H. Double Types
    I. Arithmetic Conversions
IV. Expressions - **Justin**
    A. Primary Expressions
    B. Pattern Expressions
    C. Postfix Expressions
        1. Table Referencs
        2. Function Calls
    D. Unary Operators
        1. Unary Minus Operator
    E. Casts
    F. Multiplicative Operators
    G. Additive Operators
    H. Relational Operators
    I. Equity Operators
    J. Logical AND Operator
    K. Logical OR Operator
    L. Conditional Operator
    M. Assignment Expressions
V. Declarations - **Jon**
    A. Type Specifiers
    B. Structure and Union Declarations
    C. Meaning of Declarations
    D. Table Declarators
    E. Function Declarators
    F. Pattern Declarators
    G. Statements
    H. Expression Statement
    I. Compound Statement
    J. Selection Statements
    K. Iteration Statements
VI. Program Structure - **Graham**
    A. General Structure
    B. Pattern
        1. CSS Selector
        2. Regex
    C. Action
    D. Scope
VII. Grammar

Stripped Language: AWK and LUA
Support Data Types: Double, Int, String, Set
Strongly Typed
Types inferred: 0.0, 0, “”, []
Automatic Coercion: int -> double
Operators: + - * / == = && ll []
string + string; string + int/double/set
highest precedence; returns string
set + set: concat sets
double + double: add
int + int: add
set[#] for access
Identifiers: name that is not reserved
Control Flow

```
if(expr){/*expr = 1 true, expr = 0*/

}else if(expr){

}else{

}
while(expr){

}
```

Expressions: id = expression;

```
print “”
```

General Structure

```
open “” /*Optionally opens file*/
{/*Begin*/}
[/*Pattern*/]{/*Action*/}
        [/*Subpattern*/]{/*Subaction*/}
}
[/*Pattern*/]{/*Action*/}
        [/*Subpattern*/]{/*Subaction*/}
}
{/*End*/}
```

Begin scope continues for entire pattern
Pattern scope continues for rest of program
End scope begins at start of end block and ends at closing parentheses

Functions: think of a named pattern

```
name [/*params*/]{
        return 0; //default return type
}
```

Defined in Begin and End Blocks
Patterns
[“regex”]
[“css selectors”]
Line by Line?
Access: this[#]
Open Questions
How to we handle multiple files?
Do we want to provide import statements?
What do we compile to? I suggest Java byte code since we know Java, there are good frameworks for parsing regex and xml, and it will be a lot faster than python.

Timeline

1. Determine target language (Mon)
2. Resolve open questions (Mon)
3. Fix syntax/Add + remove conventions (Mon)
4. Language Reference Manual Split (2 weeks)
    1. Introduction + Syntax
    2. Expressions
    3. Declarations
    4. Grammar