1. Introduction

The goal of this project is to build a high performance implementation of a subset of C for the purpose of creating an in memory graph database and graph analytics package. Using a subset of the ANSI C standard for the we intend to extend C to support bother RDF Graph and Property Graph structures, something we have coined the name of C+ Graph. In acknowledging that compiler design is a extraordinarily complex subject we acknowledge we will need to scale back the implementation back quite a bit since two people cannot cover all of C and the various graph structures and analytics we intend to create. The problem has been reduced to implementing the basic types and operators of C, and to implementing pointers and structures. This will enable us to effectively design C+ versions of our graph algorithms. In addition to this, we included a subset of the C preprocessor which only allows for the inclusion of header files and the definition of constants, making library-writing easier. The choice of these features bears in mind the sample programs we wish to demonstrate: data structures, algorithms, and design / output for graph structures and graph analytics.

An important aspect of this project will be the effective implementation of both RDF Graph and Property Graphs in this variant of C which we are calling C+. Because RDF Graphs are fairly popular in web ontology and a lot of NLP libraries we would like to explore their structures as well as the structures of Property Graphs and determine which algorithms are more efficient for which type. Another aspect to the project may be whether or not there is an efficient ETL process to convert from RDF Graph into Property Graph and vice versa. This is an ambitious goal but for this reason we will pick a hybrid approach where we only implement the components necessary to achieve our goal and reassess depending on how far we get in our implementation.

Please note that the ANSI C reference manual by Kernighan & Ritchie was used as a model for this document, and was frequently paraphrased. The intention of this was to preserve the definitions of the subset of original C features which we chose to implement.

2. Lexical Conventions

2.1 Tokens

The six types of tokens are identifiers, keywords, constants, string literals, operators, and other separators. All whitespace and comments are ignored except to separate tokens.

2.2 Comments

The characters /* initiate a comment, and the characters */ terminate it. Comments do not next, and can’t be included in other string or character literals.
2.3 Identifiers

An identifier is a sequence of letters and digits, beginning with a letter or _. Upper and lower case letters are distinct.

2.4 Keywords

These identifiers are reserved for use as keywords, and cannot be used otherwise:

- break
- default
- if
- static
- case
- else
- int
- switch
- char
- enum
- return
- typedef
- const
- for
- sizeof
- void
- struct
- while

2.5 Constants

There are three types of constants. Each has a data-type which will be discussed in section 4.2.

constant:

integer-constant
character-constant
enumeration-constant

2.5.1 Integer Constants

An integer constant consisting of a sequence of digits is taken to be decimal. Hexadecimal and Octal radixes will not be supported in this implementation; nor will long, short, or unsigned descriptors on integers.

2.5.2 Character Constants

A character constant is a sequence of one or more characters enclosed in single quotes, as in ‘q’. The value of a character constant with only one character is the numeric value of the characters as corresponds to the ASCII standard. Such constants may not contain ‘
’ or newlines; but these escape sequences will help to cope:

- newline: NL, \n
- Horizontal tab: HT, \t
- backspace: BS, \b
- backslash: \, \\n- Question mark: ?, \?
2.6 String Literals

A string literal is a sequence of characters surrounded by double quotes, as in “...”. A string has type “array of characters” and storage class static and is initialized with the given characters, and are immutable. Identical string literals are distinct. A null byte \0 is appended to the string so programs can find its terminus.

3. Syntax Notation

Henceforth in this manual, syntactic categories are indicated in italic, literal words and characters are indicated by typewriter style.

4. Meaning of Identifiers

TBD

5. Objects and Lvalues

Objects refer to blocks in memory, but in our implementation there are no lvalues; i.e. there is no way for the user to access an object’s virtual memory location, it is instead managed internally.

6. Conversions

TBD

7. Expressions

Precedence of expressions is the same as the order of the major subsections of this section, in decreasing order.

7.1 Primary Expressions

Primary expressions are identifiers, constants, strings, or expressions in parentheses.

```plaintext
primary-expression:
  identifier
  constant
  string
```
7.2 Postfix Expressions

The operators in postfix expressions are left-associative:

```
postfix-expression:
  primary-expression
  postfix-expression [ expression ]
  postfix-expression ( argument-expression-list )
  postfix-expression . identifier
  postfix-expression ++
  postfix-expression --
```

```
argument-expression-list:
  assignment-expression
  argument-expression-list , assignment-expression
```

7.2.1 Array References

7.2.2 Function Calls

7.2.3 Postfix Incrementation

7.3 Unary Operators

Expressions with unary operators are right-associative:

```
unary-expression:
  postfix-expression
  ++ unary-expression
  -- unary-expression
  unary-operator cast-expression
  sizeof unary-expression
  sizeof ( type-name )
```

**Unary-operator:** one of

`* - ~ !`

7.3.1 Prefix Incrementation Operators

7.4 Multiplicative Operators

The multiplicative operators `+`, `/`, and `%` are left-associative.
Multiplicative-expression:
   Multiplicative-expression * unary-expression
   Multiplicative-expression / unary-expression
   Multiplicative-expression % unary-expression

**FIXME:** need to figure out difference between “arithmetic” and “integral” types, as the C manual specifies % as integral, / and * as arithmetic.

7.5 Additive Operators

The additive operators + and - are left-associative.

   additive-expression:
      multiplicative-expression
      additive-expression + multiplicative-expression
      additive-expression - multiplicative-expression

7.6 Shift Operators

The shift operators << and >> are left-associative.

   Shift-expression:
      additive-expression
      shift-expression << additive-expression
      shift-expression >> additive-expression

7.7 Relational Operators

The relational operators are left-associative, but this fact is irrelevant.

   Relational-expression:
      shift-expression
      relational-expression < shift-expression
      relational-expression > shift-expression
      relational-expression <= shift-expression
      relational-expression >= shift-expression

7.8 Equality Operators

   equality-expression:
      relational-expression
      equality-expression == relational-expression
      equality-expression != relational-expression
7.9 Bitwise AND Operator

\[
\text{AND-expression:} \\
\text{equality-expression} \\
\text{AND-expression} \& \text{equality-expression}
\]

7.10 Bitwise Exclusive OR Operator

\[
\text{exclusive-OR-expression:} \\
\text{AND-expression} \\
\text{exclusive-OR-expression} \^ \text{AND-expression}
\]

7.11 Bitwise Inclusive OR Operator

\[
\text{inclusive-OR-expression:} \\
\text{exclusive-OR-expression} \\
\text{inclusive-OR-expression} | \text{exclusive-OR-expression}
\]

7.12 Logical AND Operator

\[
\text{logical-AND-expression:} \\
\text{inclusive-OR-expression} \\
\text{logical-AND-expression} \&\& \text{inclusive-OR-expression}
\]

7.13 Logical OR Operator

\[
\text{logical-OR-expression:} \\
\text{logical-AND-expression} \\
\text{logical-OR-expression} \| \text{logical-AND-expression}
\]

7.14 Conditional Operator

\[
\text{conditional-expression:} \\
\text{logical-OR-expression} \\
\text{logical-OR-expression} ? \text{expression} : \text{conditional-expression}
\]

7.15 Assignment Expressions

All assignment expressions are right-associative. All require lvalue as as left operand, which must be modifiable, and cannot be a function.

\[
\text{assignment-expression:} \\
\text{conditional-expression} \\
\text{unary-expression assignment-operator assignment-expression}
\]
7.16 Comma Operator

A pair of expressions is left-associative, and the result of the first is discarded.

expression:
    assignment-expression
    expression , assignment-expression

7.17 Constant Expressions

A constant expression is restricted to a sub-set of operators. They may not contain assignments, increment/decrement operators, function calls, or comma operators.

constant-expression:
    conditional-expression

8. Declarations

TBD

9. Statements

TBD

10. External Declarations

TBD

11. Scope and Linkage

TBD

12. Preprocessing

TBD
13. Grammar

14. Sample Program Translation

This dummy program declares a struct called node, makes a pointer to a new node, and initializes its edges to zero. Nothing significant, but one could imagine this being a mundane
component of a much more complex program involving graphs. This simple program is enough, however, to demonstrate many of the features we wish to implement.

```c
/* A Standard Library for input and output we will write */
#include "cpio.h"
#define EDGES 5
typedef struct node {
    int id;
    int edges[EDGES];
};
int main()
{
    struct node *nodePtr, node_1;
    nodePtr = &node_1;

    printf("initializing %d edges to our node\n", EDGES);

    for(int i = 0; i < EDGES; i++){
        (*nodePtr)->edges[i] = 0;
    } /* end for loop */
    return 0;
} /* end main function */
```

14.1 Scanner & Parser Example

Let us take the simple declaration from `main` the declaration "`struct node *nodePtr, node_1;`" and demonstrate how our scanner & parser will work with the above defined grammar rules. This statement takes the form

\[
\text{struct-declaration:} \\
\text{specifier-qualifier-list struct-declarator-list ;}
\]

equals

\[
\text{struct-or-union identifier struct-declarator , struct-declarator ;} \\
\text{[ struct ] [ node ] [ *nodePtr ] , [ node_1 ] ;}
\]

It is interesting to trace "*nodePtr" from `struct-declarator` down the hierarchy of declaration rules:

\[
\text{struct-declarator} \rightarrow \text{declarator} \rightarrow \text{pointer} \text{ direct-declarator} \rightarrow \text{pointer} \text{ identifier}
\]
We can get even more from looking at an expression instead of a declaration, making a parse-tree out of the expression “(*nodePtr)->edges[i] = 0;”
15. Graph Structures

15.1 RDF Graphs

Syntax:
Bengie is a dog.
Bonnie is a cat.
Bengie and Bonnie are friends.

The fundamental property of RDF Graphs and w3 related ontologies are the user of entities as “nodes” and relations as “edges.” This means that properties of the graph are exploded into nodes themselves and must be achieved by traversing and concatenating multiple node / edge pairs in order to achieve

15.2 Property Graphs
Conversely, property graphs are able to retain and store information directly on the node and edge itself, making it easier to retrieve values in a key-value esq

15.2 Graph Translation

16. Graph Algorithms