enum grammar

{
PIPE_ID,
RED_IN_ID,
RED_OUT_ID,
RED_APP_ID,
HERE_DOC_ID,
COMMAND_ID,
SUBSHELL_ID,
AND_ID,
OR_ID,
};

typedef struct s_entity
{

} t_entity;

typedef struct s_token
{
char *identity;
enum grammar tok;
} t_token;, void get_identity(char *input, t_token *ident)
{
int i;
ident->tok = PIPE_ID;
i = 0;

}, am working, on lexication of promt, like if the prompt is ls | cat > file, to flag ls as COMMAND_ID, and pipe as PIPE_ID, etc..., but im confused on how to set each one with his flag, please explain without coding anything for me

Certainly! Let's clarify the concept and approach of lexical analysis (lexing) for your prompt and how to assign tokens with their corresponding enum flags.

Understanding Lexical Analysis in Your Context

Lexical analysis is the process of reading an input string (like a shell command line) and breaking it down into meaningful units called **tokens**. Each token represents a syntactic category, such as a command, an operator, or a redirection symbol.

Your Goal

You want to take an input string like:

```
ls | cat > file
```

and identify each part with a specific token type from your `enum grammar`:

- `ls` → `COMMAND_ID`
- `|` → `PIPE_ID`
- `cat` → `COMMAND_ID`
- `>` → `RED_OUT_ID`
- `file` → `COMMAND_ID` (or possibly a filename token if you want to distinguish)

How to Think About Assigning Tokens

1. **Token Types Are Categories**
   Each token type in your enum represents a category of lexemes (strings). For example:
   - `PIPE_ID` for the pipe symbol `|`
   - `RED_OUT_ID` for output redirection `>`
   - `COMMAND_ID` for commands like `ls`, `cat`, etc.
2. **Identify Tokens by Matching Input Patterns**
   When you scan the input string, you look at substrings and decide which category they belong to:
   - If the substring is exactly `"|"`, assign `PIPE_ID`.
   - If the substring is exactly `">"`, assign `RED_OUT_ID`.
   - If the substring is a word that doesn't match any operator, assign `COMMAND_ID`.
3. **Order of Checks Matters**
   When scanning, first check if the substring matches any special symbols (like `|`, `>`, `<`, `&&`, `||`, etc.). If not, then treat it as a command or argument.
4. **Whitespace as Delimiters**
   Use spaces or other delimiters to separate tokens. For example, in `ls | cat > file`, spaces separate tokens.
5. **Context Awareness**
   Sometimes, the same string can mean different things depending on context. For example, `file` after `>` is a filename, but you might still treat it as a command token or create a separate token type for filenames.

Example Walkthrough (Conceptual)

- Start at the beginning of the input string.
- Read characters until you hit a delimiter (space or special symbol).
- Extract the substring:
  - If substring is `"|"`, assign `PIPE_ID`.
  - If substring is `">"`, assign `RED_OUT_ID`.
  - Otherwise, assign `COMMAND_ID`.
- Move to the next substring and repeat.

Summary

- **Scan input left to right.**
- **Split input into substrings by spaces and special characters.**
- **Match substrings against known operators first (pipe, redirections, logical AND/OR).**
- **If no match, classify as `COMMAND_ID`.**
- **Assign the corresponding enum value to each token.**

This approach allows you to flag each part of the prompt correctly during lexical analysis.

If you want, I can also help you design the logic or data structures to implement this clearly. Just let me know!

the issue im having, is how each one is going to get his flag alone,

Great question! The core of your confusion seems to be **how to assign the correct enum token (`tok`) to each piece of the input individually** during lexing. Let me explain the concept clearly and step-by-step without writing code, so you can understand the logic behind it.

How Each Token Gets Its Own Flag (Enum Value)

1. Token Extraction (Splitting the Input)

- First, you **split the input string into smaller parts**, called *lexemes* or *token strings*. For example, from the input:

  ```
  ls | cat > file
  ```

  You get these lexemes:

  - `"ls"`
  - `"|"`

- "cat"
- ">"
- "file"

- Each lexeme is a candidate for a token.

2. Token Classification (Assigning the Enum Flag)

- For each lexeme, you **check what it represents** by comparing it to known symbols or patterns.
- You can think of this as a **decision tree or a set of if-else checks**:
  - **Is the lexeme exactly `"|"`?**
    → Yes → assign `PIPE_ID`
  - **Is the lexeme exactly `">"`?**
    → Yes → assign `RED_OUT_ID`
  - **Is the lexeme exactly `"<"`?**
    → Yes → assign `RED_IN_ID`
  - **Is the lexeme exactly `"&&"`?**
    → Yes → assign `AND_ID`
  - **Is the lexeme exactly `"||"`?**
    → Yes → assign `OR_ID`
  - **Is the lexeme a word that is not any of the above?**
    → Yes → assign `COMMAND_ID`
- This way, each lexeme is **individually checked** and assigned the correct token type.

3. Storing the Token

- After deciding the token type, you create a `t_token` instance for that lexeme:
  - `identity` points to the lexeme string (e.g., `"ls"`, `"|"`, etc.)
  - `tok` is set to the enum value you determined (e.g., `COMMAND_ID`, `PIPE_ID`)
- You then store or process this token for the next stages (parsing, execution, etc.).

4. Why This Works

- Because you **process each lexeme separately**, each token gets its own enum flag.
- The key is the **classification step**: a simple check of the lexeme against known symbols or patterns.

Visualizing the Process

Lexeme	Check Against	Resulting Token Type (`tok`)
`"ls"`	Not a special symbol	`COMMAND_ID`
`"|"`	Matches pipe symbol	`PIPE_ID`
`"cat"`	Not a special symbol	`COMMAND_ID`
`">"`	Matches redirection	`RED_OUT_ID`
`"file"`	Not a special symbol	`COMMAND_ID` (or filename)

Summary

- **Split input into lexemes (tokens).**
- **For each lexeme, check its value against known symbols.**
- **Assign the corresponding enum flag based on the match.**
- **Store each token with its lexeme and enum flag.**

This method ensures **each token gets its own flag independently**.

If you want, I can help you design a flowchart or pseudocode for this classification step to make it even clearer!

so how is going to work , in the manner of my struct, where i store the char * and its enum, isnt ecah time i store it overwrite the last one

The core issue you're describing-how to store multiple tokens each with their own `char *` identity and enum flag without overwriting previous ones-comes down to **how you organize your data structures and memory**.

Why Storing Tokens in a Single `t_token` Variable Overwrites Previous Tokens

If you have just one variable of type `t_token` (e.g., `t_token ident;`) and you keep assigning to it as you scan the input, then yes, each new assignment will overwrite the previous one, because it's the same memory location.

How to Store Multiple Tokens Without Overwriting

1. Use a Collection (Array or Linked List) of `t_token`

Instead of a single `t_token`, you need a **container** that can hold many tokens, for example:

- An array: `t_token tokens[MAX_TOKENS];`
- A dynamically allocated array (resizable)
- A linked list of `t_token` nodes

Each element in this container holds one token (with its own `identity` and `tok`).

2. Copy or Allocate Memory for Each Token's `identity`

- The `identity` field is a `char *` pointer to the token string.
- You must **allocate separate memory** for each token's string or store a pointer to a unique substring.
- If you just point `identity` to the same input buffer without copying, all tokens might end up pointing to the same buffer location that changes.

3. Example Conceptual Flow

- Initialize an array or list of `t_token`.
- For each lexeme you extract:
  - Allocate memory for the lexeme string or use a substring pointer that won't be overwritten.
  - Assign the string pointer to `identity`.
  - Assign the correct enum flag to `tok`.
  - Store the `t_token` in the next slot of your array or add it to your list.
- Move on to the next lexeme.

This way, each token is stored separately, and no token overwrites another.

Summary

Problem	Solution
Single `t_token` variable reused	Use an array or linked list of `t_token`
`identity` pointer overwritten	Allocate/copy string for each token's identity
Tokens overwrite each other	Store each token in a separate array/list element

Additional Notes

- You can define a struct like:

```
typedef struct s_token {
    char *identity;
    enum grammar tok;
} t_token;

t_token tokens[MAX_TOKENS];  // Array to hold multiple tokens
int token_count = 0;
```

- When you identify a token, copy the substring into a newly allocated string (e.g., using `strdup`) and assign it to `tokens[token_count].identity`, set `tok`, then increment `token_count`.

This approach ensures **each token is stored independently with its own string and enum flag**, preventing overwriting.

⁂

elaborate on the matter of a linked list

A linked list is an ideal data structure to store multiple tokens dynamically during lexical analysis without overwriting previous ones. Here's how it works and why it fits your case:

What Is a Linked List?

- A linked list is a chain of **nodes**, where each node holds:
    - **Data** (in your case, a token: `char *identity` and `enum grammar tok`)
    - A **pointer** to the next node in the list
- The list starts from a **head** pointer referencing the first node
- The last node's next pointer is `NULL`, marking the end of the list

Why Use a Linked List for Tokens?

- You don't need to know in advance how many tokens you will have (unlike a fixed-size array)
- You can **dynamically add tokens** as you parse the input, creating a new node for each token
- Each token is stored in its own node, so **no overwriting occurs**
- Easy to traverse later for parsing or execution

How to Structure a Linked List Node for Your Tokens

You can define a node struct like this (conceptually):

```
typedef struct s_token_node {
    char *identity;          // token string (e.g. "ls", "|", ">")
    enum grammar tok;        // token type (e.g. COMMAND_ID, PIPE_ID)
    struct s_token_node *next; // pointer to next token node
} t_token_node;
```

- Each node holds one token's data and a pointer to the next token
- The linked list is a sequence of these nodes connected via `next`

How It Works Conceptually When Lexing

1. **Start with an empty list:** `t_token_node *head = NULL;`
2. **For each lexeme you extract from the input:**
   - Allocate a new node (`malloc`)
   - Copy or assign the lexeme string to `identity`
   - Determine and assign the correct enum flag to `tok`
   - Set the new node's `next` pointer to `NULL`
3. **Add the new node to the end of the list:**
   - If the list is empty (`head == NULL`), set `head` to the new node
   - Otherwise, traverse the list to the last node and set its `next` pointer to the new node
4. **Repeat for all tokens**

Benefits and Practical Notes

- Each token is stored separately in its own node, so no overwriting happens
- You can easily traverse the list later to process tokens in order
- Memory is allocated dynamically, so the list can grow as needed
- You can insert tokens anywhere if needed (not just at the end)

Summary

Concept	Explanation
Node	Holds one token's string, enum flag, and pointer to next node
Linked List	Chain of nodes connected via `next` pointers
Adding Tokens	Create a new node per token and link it at the end

Concept	Explanation
Storage	Each token stored independently, no overwriting
Flexibility	Dynamic size, easy insertion and traversal

Using a linked list for your tokens lets you build a dynamic, ordered collection of tokens during lexing, perfectly matching your need to assign and store each token's identity and enum flag separately.

⁂

t_token *add_identity(char *content, enum)

{
t_token *new_node;

```
new_node = malloc(sizeof(t_token));
if (!new_node)
        exit(F);
if (new_node)
{
        new_node->identity = content;
        cpy_identity(new_node->identity, content);
        new_node->tok = enum;
        new_node->next = NULL;
}
return (new_node);
```

}, how can i send the enum to this function

To send an enum value to your function `add_identity`, you simply declare the parameter with the enum type and pass one of the enum constants when calling the function.

Given your enum:

```
enum grammar {
    PIPE_ID,
    RED_IN_ID,
    RED_OUT_ID,
    RED_APP_ID,
    HERE_DOC_ID,
    COMMAND_ID,
    SUBSHELL_ID,
    AND_ID,
    OR_ID,
};
```

You can declare your function like this (fixing your snippet):

```c
t_token *add_identity(char *content, enum grammar tok)
{
    t_token *new_node;

    new_node = malloc(sizeof(t_token));
    if (!new_node)
        exit(EXIT_FAILURE);
    new_node->identity = strdup(content);  // copy string safely
    new_node->tok = tok;
    new_node->next = NULL;
    return new_node;
}
```

Then, when you call `add_identity`, pass the enum constant as the second argument, for example:

```c
t_token *token1 = add_identity("ls", COMMAND_ID);
t_token *token2 = add_identity("|", PIPE_ID);
t_token *token3 = add_identity(">", RED_OUT_ID);
```

Key points:

- The second parameter of `add_identity` should be of type `enum grammar` (or your enum type).
- Pass the enum constant directly when calling the function.
- Inside the function, assign the enum parameter to the token's `tok` field.
- Use `strdup` (or equivalent) to copy the string content to avoid pointer aliasing issues.
- Make sure to `#include <stdlib.h>` and `<string.h>` for `malloc` and `strdup`.

This approach is standard in C for passing enum values as function arguments[1] [2] [3].

⁂

what comes after lexing and setting all the tokens with their propriate IDs, dont code anything just assist me

After lexing and assigning the appropriate token IDs to each lexeme, the next phase is **Syntax Analysis (Parsing)**.

What Comes After Lexing?

1. Syntax Analysis (Parsing)

- The parser takes the stream of tokens produced by the lexer as input.
- It checks whether the sequence of tokens conforms to the grammatical rules of the language (syntax).
- The parser builds a **parse tree** or **abstract syntax tree (AST)** representing the hierarchical syntactic structure of the input.
- For example, it verifies that commands, pipes, redirections, and logical operators are arranged in a valid way according to the shell grammar.
- If the token sequence violates syntax rules, the parser reports syntax errors.

2. Semantic Analysis

- After parsing, semantic analysis ensures the meaning of the token sequence is valid.
- It checks for semantic errors such as type mismatches, undeclared identifiers, or invalid command usage.
- It may also involve symbol table management, tracking identifiers and their attributes.

3. Intermediate Code Generation and Beyond

- Once syntax and semantics are validated, the compiler or interpreter may generate an intermediate representation of the command.
- Later stages include optimization and final code generation or execution.

Summary of Phases After Lexing

Phase	Role
Syntax Analysis	Check token order against grammar; build parse tree or AST
Semantic Analysis	Validate meaning and context; manage symbol tables and check for semantic errors
Intermediate Code Generation	Create an abstract, machine-independent representation of the input
Code Optimization	Improve intermediate code for efficiency
Code Generation	Produce final executable code or commands

In your shell prompt lexer context, after lexing tokens like commands, pipes, and redirections, the **parser** will analyze the token sequence to ensure it forms valid shell command syntax before any execution or further processing.

This progression from lexing to parsing is standard in compiler and interpreter design[4] [5] [6] [7].

⁂

so technically now, i have a linked list with all the entities with their ID, i should now build a parser, that check the context, for example if we meet command we should check if the argument is placed infront of it, and so...

Exactly! You've got the right idea.

What to Do Next: Building the Parser

1. Input to the Parser

- Your parser will take the **linked list of tokens** (each with its `identity` string and `tok` enum) produced by your lexer.

2. Purpose of the Parser

- The parser's job is to **understand the structure and meaning of the token sequence** according to the shell's grammar rules.
- It checks **contextual correctness** - not just what each token is, but how tokens relate to each other.

3. Contextual Checks You Mentioned

- **Commands and arguments:**
  - After a `COMMAND_ID` token, it's typical to expect zero or more argument tokens (which might also be `COMMAND_ID` or a separate argument token type if you define one).
- **Operators and redirections:**
  - After a pipe (`PIPE_ID`), you expect another command or subshell.
  - After a redirection token (`RED_OUT_ID`, `RED_IN_ID`, etc.), you expect a filename token.
- **Logical operators (`AND_ID`, `OR_ID`):**
  - They connect commands or command groups and require valid commands on both sides.
- **Subshells:**
  - Tokens representing subshells (`SUBSHELL_ID`) should be parsed as grouped commands.

4. Building the Parser Logic

- **Traverse your linked list of tokens sequentially.**
- **Use state or recursive functions** to:
  - Validate the order and combination of tokens.
  - Group tokens into higher-level structures (e.g., command with arguments, pipelines, conditional commands).
- **Detect syntax errors** if tokens appear in invalid sequences.
- **Build a data structure (like an Abstract Syntax Tree)** representing the command hierarchy and relationships.

5. Why This Matters

- The parser transforms a flat token list into a structured representation.
- This structure is essential for:
  - Correct execution of commands.
  - Handling complex constructs like pipes, redirections, and conditionals.
  - Providing meaningful error messages.

Summary

Step	Description
Lexer	Produces tokens with types (`COMMAND_ID`, `PIPE_ID`, etc.)
Parser	Checks token order and context; builds command structure
Contextual validation	Ensures commands have proper arguments, operators are correctly placed, etc.
Syntax tree building	Organizes tokens into a hierarchical structure for execution

If you want, I can help you outline how to design such a parser or explain common parsing strategies (recursive descent, state machine, etc.) tailored to shell command syntax!

1. https://www.w3schools.com/c/c_enums.php
2. https://stackoverflow.com/questions/4654655/how-to-pass-enum-as-function-argument-in-c
3. https://www.programiz.com/c-programming/c-enumeration
4. https://unstop.com/blog/phases-of-a-compiler
5. http://www.icet.ac.in/Uploads/Downloads/M1.pdf
6. https://byjus.com/gate/phases-of-complier-notes/
7. https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm