Project Task C Parser

Implement a parser for `c4`:

- For expressions (§6.5) we handle *identifier*, *constant*, *string-literal*, parenthesized expression, `[]`, function call, `.`, `->`, `sizeof`, `&` (unary), `*` (unary), `-` (unary), `!`, `*` (binary), `+` (binary), `-` (binary), `<`, `==`, `!=`, `&&`, `||`, `?:` and `=`. For all other expressions only the chain productions ($A \to B$) are used.
- At declarations (§6.7) we only consider *init-declarator-list* with at most one *init-declarator* without *initializer*. The only *declaration-specifiers* and *specifier-qualifier-list* is *type-specifier*. *type-specifier* is restricted to `void`, `char`, `int` and *struct-or-union-specifier*. The latter is only `struct` without *type-qualifier*s and bit fields. *declarator* and *direct-declarator* are *pointer* (without *type-qualifier-list*), *identifier*, parenthesized declarator and function declarator with *parameter-type-list*. *parameter-type-list* is only *parameter-list* without ellipses (`...`). All productions for *parameter-declaration* are considered. The productions for *abstract-declarator* and *direct-abstract-declarator* are restricted accordingly:

```
declaration:
    type-specifier declarator_opt ;

type-specifier:
    void
    char
    int
    struct-or-union-specifier

struct-or-union-specifier:
    struct identifier_opt { struct-declaration-list }
    struct identifier

struct-declaration-list:
    declaration
    struct-declaration-list declaration

declarator:
    pointer_opt direct-declarator

direct-declarator:
    identifier
    ( declarator )
    direct-declarator ( parameter-list )

pointer:
    *
    * pointer

parameter-list:
    parameter-declaration
    parameter-list , parameter-declaration

parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declarator_opt

type-name:
    type-specifier abstract-declarator_opt
```

```
abstract-declarator:
    pointer
    pointer_opt direct-abstract-declarator

direct-abstract-declarator:

    ( abstract-declarator )
    direct-abstract-declarator_opt ( parameter-list )
```

- The considered *statement*s (§6.8) are *labeled-statement* with an identifier, *compound-statement*, *expression-statement* (both expression and null statements), *selection-statement* with `if` and `if-else`, *iteration-statement* with `while` and every *jump-statement*.
- The root are external definitions (§6.9), which are handled fully except for *declaration-list* in *function-definition*.

- Note that the subset of `c4` is carefully chosen such that you are able to perform the semantic analysis independently *after* parsing as `c4` does not support `typedef`.

For parsing your compiler will be invoked with `c4 --parse test.c`. The parser must reject all words that are not derivable from the full grammar. The parser must accept all correct programs according to the restricted grammar. As error locations, choose the location of (the start of) the first token that makes the token sequence not a word of the grammar.[1] If the error is caused by a missing token at the end of the token sequence, report the location of the end of the file, i.e. the location after the last character of the file.

Remarks and hints:

- The grammar as given is not suitable for LL parsing in some places. For these places, adjust the grammar accordingly and/or use precedence climbing.
- Don't repeat yourself! Factorize common operations into helper functions.
- The grammar is ambiguous for *selection-statement*s. How is this ambiguity resolved in the language standard? How can this be treated in the implemenation of the parser?
- How to implement the k-lookahead capability in your lexer/parser?
- It is not yet required to construct an abstract syntax tree, but will be necessary for the next parts of the project. What classes and class hierarchy for AST nodes do you need, e.g. `Expression`, `BinaryExpression`?
- You may choose to delay some checks required to ensure that the input is a word of the grammar to the later semantic analysis stage. Once you implement it, the semantic analysis will run together with your parser in the tests.
- The soft deadline for this milestone is 2025-11-23.
- Keep it simple!

[1] While this is not necessary the most helpful error location for developers using the compiler, it is better for automated testing.