PROJECT

For the project, you will be extending the basic parser you created in assignment3. Specifically, you will start by categorizing tokens into several broad categories along with storing the type for identifiers and literals. Then you will find all conditional expressions within the code and then check for certain kinds of errors in the input files.

For this project we have provided a new parserClasses.h header file. You are to implement all functions in the parserClasses.h file that do not have inline implementations in your existing parserClasses.cpp file as outlined in the following sections.

Part1: Extending the Token Class and Categorizing Tokens (Week of November 16th)

For the project, the Token class has been extended with new functionality (see parserClasses.h). Some of this new functionality will require some member functions for the Token class to be updated. To begin with you should provide the new implementations for the constructors and other new Token class member functions.

Once that is done, you will need to complete the logic for determining token type. Token type determination should be done after all tokens have been found by calling the new TokenList member function:

```
//find token details and type and update token.
//May require examining properties of neighbouring tokens
void findAndSetTokenDetails(Token *token);
```

Once you have found and set all the properties for the tokens you should create new TokenLists in project.cpp for each category of tokens for testing purposes.

Keywords: Keywords[1] are any reserved word in the language (e.g. true, xor, signal) they do NOT include types included through libraries (e.g. std_logic, std_logic_vector)

Categories: Tokens are broken down into five major categories: identifiers, operators, literals, comment bodies, and other.

Example:

```
signal a : std_logic;
signal b : std_logic;
a <= b and '1'; -- a comment
```

`a`, `signal`, `std_logic`, and `b` are identifiers.
`<=` and `and` are operators.
`'1'` is a literal. (As would be `5` and `"01101"` or `X"123ABC"`)
`:` , `;` and `--` are of type other.
` a comment` is a comment body.
Additionally, `and` and `signal` are keywords.

1 A list of VHDL keywords can be found here: www.csee.umbc.edu/portal/help/VHDL/reserved.html

Identifiers and Literals:

For identifiers (that are for variables and signals) and literals you also need to be able to identify the type of the variable and the width (if it is a vector type).

```
signal <name> : <type> := <default value>; -- or without default value
variable <name> : <type> := <default value>;-- or without default value
```

for vectors, type will be of form:

```
<type>(# downto/to #)
```

You do NOT have to support identifying the type for any signals/variables outside of these cases.

RemoveToken / RemoveComments / removeTokensOfType:

In the updated version of parserClasses.h there is an additional remove token function:

```
int removeTokensOfType(TokenList &tokenList, tokenType type);
```

for removing all tokens of a certain category. You need to provide an implementation for this function, and if you did not complete the challenge task from assignment3, you also need to complete those functions for the project as well. If you implement the removeComments function after you implement categorizing tokens, you may find the function easier to implement.

Part2: Finding conditional expressions inside when/else and if /elsif /else constructs (Week of November 16th)

For this part of the project we want you to find all conditional expressions inside of when/else structures and if/elsif/else structures:

```
if <conditional expression> then
elsif <conditional expression> then
when <conditional expression> else ...
```

Finding all conditional expressions should be implemented in a new function `findAllConditionalExpressions`, in parserClasses.h. The function creates a new TokenList and appends tokens that are part of a conditional expression to the list. At the end of each conditional expression a new line token is to be inserted.

Example:

```
if (a = true) then
…
elsif b /= c or a = '1' then
```

The list should include: `(`, `a`, `=`, `true`, `)`, `\n`, `b`, `/=`, `c`, `or`, `a`, `=`, `'1'`, `\n`

Part3: Error Checking (Week of November 30th)

Checking for complete (if then / elsif then / else / end if) pairing

In VHDL, every if statement must end with an end if. It may have multiple elsifs in between, and it may also have an else case. For every “if” or “elsif” there must be a corresponding “then”.

Checking for type matching in conditional expressions

When you compare two identifiers, you need to ensure that they are the same type and, if bit vectors, the same width. When comparing against a literal against an identifier you have to check to see if the literal's properties match the identifiers.

Example:

```
signal a : std_logic;
signal b : std_logic_vector(0 to 1);
…
if a = “0011” then -- type mismatch
if b = “0011” then -- width mismatch
```

Parts 1 and 2 will be marked by an autochecker whereas part3 will be marked during your in lab demo. For this part you have full discretion on how you choose to implement the required functionality.

You should be able to return statistics on any input you analyze in both a user select able verbose and non-verbose mode. The output should be clear and easy to understand for the user in either mode:

In non-verbose mode (at least):

- The number of tokens
- The number of conditional expressions
- The number of missing “end if”s
- The number of missing “then”s
- The number of type mismatches
- The number of width mismatches

In verbose mode (at least):

- All of the non-verbose counts.
- For all error cases print out the line(s) on which the error occurs along with its type
- Any additional details you wish to include

Bonus Tasks:

You may add bonus features to your program. Possible bonus features you might want to consider:

- Detecting additional types of errors in conditional expressions, (e.g. missing brackets, operators, missing semicolons, etc.)
- Supporting user defined types (new types made declared with the type keyword)

Building your project: For the project, we have provided you with a Makefile to help you build your project as well as to help you prepare your submission. To build your submission, simply type:

```
$ make
```

in the directory where the Makefile exists.

INSTRUCTIONS FOR SUBMITTING YOUR CODE

- For the project you will be submitting only the c++ files (i.e. "*.cpp" and "*.h") for your project
- To submit your project make sure your project.cpp, parserClasses.cpp and parserClasses.h file are in the same directory as the new provided makefile and run:

  ```
  $ make tar
  ```
- **upload the resulting tar.gz file to the submission server.**