Compiling Techniques (ECOTE)

final project

Semester: Summer 2022
Author: Laura Ploch, 300176
Subject: "XML to C++"

Task:
Write a program which reads XML file, generates code of C++ classes according to the XML file structure and code that creates objects of those classes with attribute values read from the XML file.

Contents:

1. General overview and assumptions
2. Functional requirements
3. Implementation
3.1. General architecture
3.2. Data structures
3.3. Module descriptions
3.4. Input/output description
4. Functional test cases

1. General overview and assumptions

Program purpose is to generate C++ classes from XML file. The supported types are C++ built-in types and std::string. The classes declared in XML file will consist of the name of the class instance and at least one attribute declaration. The output will consist of header files and classes named in order of discovery (Class1, class1.h, Class2, class2.h, …) as well as the main.cpp file with declarations of discovered instances of classes.

2. Functional requirements

1. Input is an XML file.
2. Output consists of header files for each created class and a main.cpp file with objects declarations.
3. If no class nor container declarations are detected in the input file, the output is an empty main function in main.cpp file.
4. The input file must be a valid XML file, in particular containing start and end tags (root element that is the parent of all other elements).
5. Each class declaration must have a name of the instance of the class and at least one attribute declaration.
6. Each attribute name must be unique.
7. The order of the attributes does matter.
8. If declared object's attributes are different from the attributes of each already existing class, a new class is generated.
9. Output classes contain default constructor, constructor with parameters and getters and setters for each attribute.

3. Implementation

Implementation will be done in Python 3.10.0.

General architecture

1. Scan the input and generate tokens
2. Parse the tokens and create classes and instances of classes as designed python objects
3. Generate headers and main file on the basis of created classes and their instances

Data structures

- Tokenizer
  A class responsible for scanning the input XML file and producing tokens. It holds *tokens*, *forbiddenASCII* - ASCII values of forbidden characters, *operators* and *linesScanned* to keep track of the number of the line currently scanned in order to provide this information when displaying an error. It also holds *methods* like: *scan, readTag, readString, generateForbiddenAscii* and *errorChar* that will be described in more detail later.

- Parser
  A class responsible for parsing the tokens scanned and listed by the *Tokenizer,* checking the syntax and correctness of the declarations, and creating classes and instances of classes. Its parameters are *t – Tokenizer*, *c – Code_Generator, tokens* and lists of *classes* and *instances*, which hold objects of type: *Cpp_class* and *Cpp_object* specified below. The parser's methods are: *do* – short function, responsible for the main operation of the program (calling Tokenizer, calling parsing function and calling code generation), *parseTokens*, *readElement, addClass, getId, getSameClass* and *addInstance.*

In classes.py:

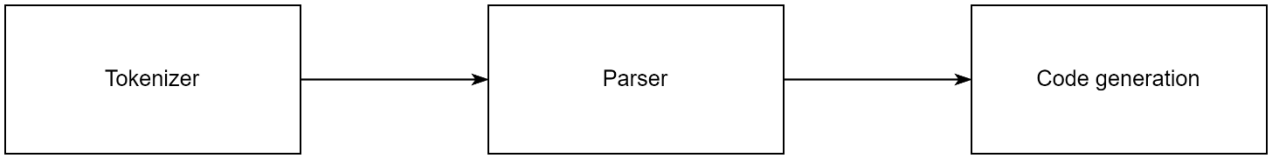
- Cpp_class
  A class used for holding information about each discovered class. List *attributes* holds all discovered attributes (all are unique) and *pointers* hold all discovered pointers names. The class has an overloaded comparison operator *__eq__* used during parsing to compare the discovered class to the classes that are already kept in parser's internal list of classes. Another method *add_attribute* is used to compare provided attribute to the ones already held by the class, in order to add only unique names of attributes.

- Cpp_object
  A class representing objects of cpp_classes that will be generated in the main file. Containing *baseClassId* – id of the class it is an instance of, *name* – name of the instance, read from the XML element, and *attributes* dictionary for keeping discovered attributes with their values. It also has an overloaded comparison operator.

In codegenerator.py:

- Code_Generator
  Class responsible for generating output files. Has methods: *generateHeaders, generateMain* and *generateEmptyMain* – for the situation when no classes are declared.

Module descriptions



Tokens descriptions:

program	<prog_id> prog_body </prog_id>
prog_id	IDENTIFIER ex) root
prog_body	{class_declaration | container_declaration}
class_declaration	<object_id attr_declarations> class_body </object_id>
class_declaration	<object_id attr_declarations/>
object_id	IDENTIFIER
attr_declaration	attr_id=string_value
attr_id	IDENTIFIER
String_value	STRING
class_body	<pointer_id> class_declaration </pointer_id>
pointer_id	IDENTIFIER
container_declaration	<container_id> class_declaration </container_id>

IDENTIFIER token will begin with a letter and be followed by letters or numbers. IDENTIFIERS are case sensitive and can't be C++ keywords.

STRING token should always begin and end with double quotation mark ". It cannot contain extra double-quotation marks inside.

OPERATORS: < > / = "

1. Tokenizer

In function *scan* the tokenizer reads the input character by character and identifies the characters as either one of operators: < > / = ", or it detects a letter and calls either *readTag* function to scan the tag or *readString* function which read a string until it encounters a double quotation mark '"'. *ReadString* function is called if a tag has already been read and scanner detected and read an attribute.

Each token is appended to the token list with appropriate type. 0 – operator, 1 – tag, 2 – attribute and 3 – stringvalue. The scanner ignores the white spaces outside of strings. It initially checks the correctness of the input. Additionally, when it encounters a new line character '\n' it increases the *linesScanned* value to be able to provide the line number in case of an error. When the scanning is completed, the tokens are passed to the parser.

2. Parser

The main part of the program. Calls the remaining parts (tokenizer, code generation). Parses tokens one by one, popping each read element from the list after it has been successfully processed. It first checks the root element – if both beginning and ending tag are present, if the closing tag matches the opening tag. After that, it enters a *while loop* which ends when there are no more tokens to parse (all of them have been popped). Inside, it always looks for the opening operator "<" first and then, if at least one attribute declaration is present, it calls *readElement* function to read the whole class declaration.

ReadElement function reads the name of the instance and declarations of attributes. Holds them in temporary *tempClass* and *tempObject*. After it reaches the end of the declaration – "/>" operator, it uses *addClass* function to add the *tempClass* to the list of classes (or not, if such class already exists), which also returns the pointer to the added class, or the one already existing (using *getSameClass* function). This is later used to obtain ID of the class in the list of classes (*getId* function) to assign it to the *baseClassId* of the *tempObject.* Then the object is added to the parser's list of instances using *addInstance* function, which throws an error if the name of the instance has already been declared before.

When all tokens have been processed, the lists of classes and instances are passed to the *generateFiles* function of the Code_Generator class.

3. Code_Generator

Converts provided *Cpp_classes* into their C++ equivalents in separate header files and *Cpp_objects* into declarations of instances of the classes in the main.cpp file. If the sizes of the lists are 0, then the code generator creates empty main function in main.cpp file.

Input/output description

Input file is an XML file. The file name is provided as a command line parameter. The rules for creation of the XML file are mentioned in functional requirements.

Output files are created in the same directory as the location of the program. The output files will consist of separate header files for each class declaration and main.cpp file for declarations of instances of the classes. The header files will be included in the main file.

4. Functional test cases

For brevity, <root> elements are omitted in some example inputs.

1. Invalid XML file - no end tag
   Example input: `<root>`
   Expected output: Error during parsing.

2. Invalid XML file – unmatched tags
   Example input:
   ```
   <root>
   </rot>
   ```
   Expected output: Error during parsing. The parent element must have a matching closing tag.

3. XML file containing only start and end tag with no tokenizable content
   Example input:
   ```
   <root>
   </root>
   ```
   Expected output: Empty main function in main.cpp file

4. Correctly declared instance of a class
   Example input: `<newman name = "John" surname = "Newman"/>`
   Expected output: Class1 class generated in class1.h header and instance of Class1 newman

5. Unexpected operator
   Example input: `<= newman name="John" surname="Newman"/>`
   Expected output: Error during scanning. Unexpected operator.

6. Unexpected character
   Example input:
   ```
   <root> word
        <newman name="John" surname="Newman"/>
   </root>
   ```
   Expected output: Error during scanning. Unexpected character “w”.

7. Declarations of three instances and two different classes.
   Example input:

```
<root>
    <newman name = "John" surname = "Newman"/>
    <daniel name = "Daniel" cat = "Puszek"/>
    <jerry name = "Jerry" surname = "Nowak"/>
</root>
```

   Expected output: Two header files with definitions of Class1 and Class2. Main file with declarations Class1 newman, Class2 daniel and Class1 jerry

class1.h

```
#include <string>
class Class1 {
    private:
        std::string name;
        std::string surname;

    public:
        Class1() { };
        Class1(std::string name, std::string surname) {
            this->name = name;
            this->surname = surname;
        }

        std::string getName() {
            return name;
        }
        void setName(std::string name) {
            this->name = name;
        }
        std::string getSurname() {
            return surname;
        }
        void setSurname(std::string surname) {
            this->surname = surname;
        }
};
```

class2.h – analogically, with parameters *name* and *cat*

main.cpp:

```
#include <string>
#include <Class1.h>
#include <Class2.h>
#include <Class3.h>

int main() {
    Class1 new1man("John", "Newman");
    Class2 daniel("Daniel", "Puszek");
    Class3 jerry("Jerry", "Nowak");
};
```

8. ‘=’ operator missing in attribute declaration
   Example input:
   ```
   <root>
        <newman name "John" surname="Newman"/>
   </root>
   ```
   Expected output: Error during scanning. '=' missing in attribute declaration.

9. Attribute names are not unique
   Example input: `<newman name = "John" name = "Newman"/>`
   Expected output: Error during parsing. Attribute name repeated in the same element.

10. Tag starting with a number
    Example input: `<1newman name = "John" surname = "Newman"/>`
    Expected output: Error during scanning, in line 2. Unexpected character “1”.

11. Instance name (tag) and attribute name declared with a number in name
    Example input: `<new1man name2 = "John" surn3ame = "Newman"/>`
    Expected output: Header file(s) created succesfully. Main file created succesfully.

12. Missing ‘/>’ from class declaration
    Example input: `<newman name = "John" surname = "Newman">`
    Expected output: Error during parsing. Missing closing tag '/>'.