Sabancı University
Faculty of Engineering and Natural Sciences

CS305 Programming Languages

Homework 4

Due: 08-04-2019

1 Introduction

In this homework you will implement a semantic checker and a simple translator for "CSML" programming language for which you have implemented a parser in the last assignment. A bison/flex implementation is provided to you, so even if you were not able to complete the previous assignment you can continue from the provided parser implementation. However, there can be modifications needed on these files to implement the required features of the semantic checker/translator. You will have to add the construction of an abstract syntax tree and perform the semantic checks on this tree.

For the CSML syntax, please refer to the previous assignment where it was explained in detail. The semantic rules that you have to check are listed below. Here is a grammatically correct but semantically incorrect CSML program which will be used to explain several semantic errors:

```
 1  <course code="CS192" name='Programming Languages' type="Lecture">
 2      <class section="0" instructor='Husnu Yenigun' crn=20258 capacity=60>
 3          <meeting day=R start=08:40 end=10:30>
 4          </meeting>
 5          <meeting start=08:40 end=09:30 day=F/>
 6      </course>
 7  </class>
 8  <course code="CS301R" name="Algorithms-Recitation" type="Recitation">
 9      <class section="0" instructor='Husnu Yenigun' crn=20257>
10          <meeting start=17:40 day=M end=18:40/>
11      </class>
12  </constraint>
13  <constraint>
14      <item code="CS305"/>
15      <item crn=20257/>
16   </constraint>
```

Figure 1: An example CSML program with a lot of semantic errors

2 Semantic Rules

Your semantic checker should perform an analysis for the following semantic rules. For each violation of these rules, your semantic analyzer must print out an error message. Note that, after printing out an error message, your semantic analyzer must not terminate and keep working to find other violations, if there exists any.

SR1: **There must be exactly one occurrence of each attribute in a course element (15 points)**

Each course section should have exactly one course `code` attribute, one course `name` attribute and one course `type` attribute.

These attributes can be given in any order.

The following is a correct class element:

```
<course code="CS305"  name='Programming Languages' type="Lecture">
```

whereas the following is incorrect as there are three `code` attributes and no `name` attribute:

```
<
course code="CS305" type="Lecture" code="CS306" code="CS307"
>
```

When there are more than attribute of the same type in the attribute list of a course element, or when there is a missing attribute in the attribute list of a course element, you must produce an error message in the following format:

`ERROR: course element at line` **X** `has` **Y** `occurrences of` **type**

where **X** is the line number in the input at which the corresponding `tCOURSE` token appears, **Y** is the number of occurrences of the corresponding attribute, and **type** is the problematic attribute.

For example, the erroneous course element given above should produce the following error messages:

```
ERROR: course element at line 2 has 3 occurrences of code
ERROR: course element at line 2 has 0 occurrences of name
```

SR2: **There must be exactly one occurrence of each attribute in a class element (15 points)**

Each class element should have exactly one `section` attribute, one `instructor` attribute, one `crn` attribute, and one `capacity` attribute.

These attributes can be given in any order.

The following is a correct class element:

```
<class section="0" instructor="Husnu Yenigun" crn=20759 capacity=192>
```

whereas the following is incorrect as there are three `section` attributes, two `crn` attributes and no `capacity` attribute:

```
<
class
crn=1 section="0" crn=1 section="1" section="2"
instructor="Nice Newday"
>
```

When there are more than attribute of the same type in the attribute list of a class element, or when there is a missing attribute in the attribute list of a class element, you must produce an error message in the following format:

`ERROR: class element at line` **X** `has` **Y** `occurrences of` **type**

where **X** is the line number in the input at which the corresponding `tCLASS` token appears, **Y** is the number of occurrences of the corresponding attribute, and **type** is the problematic attribute.

For example, the erroneous class element given above should produce the following error messages:

```
ERROR: class element at line 2 has 3 occurrences of section
ERROR: class element at line 2 has 2 occurrences of crn
ERROR: class element at line 2 has 0 occurrences of capacity
```

SR3: **There must be exactly one occurrence of each attribute in a meeting element (15 points)**

Each meeting element should have exactly one `start` attribute, one `end` attribute, and one `day` attribute.

These attributes can be given in any order.

The following is a correct class element:

```
<meeting start=10:40 end=12:40 day=M />
```

whereas the following is incorrect as there are two `start` attributes and no `day` attribute:

```
<
 meeting
   start=10:40
   end=12:40  start=11:40
/>
```

When there are more than attribute of the same type in the attribute list of a meeting element, or when there is a missing attribute in the attribute list of a meeting element, you must produce an error message in the following format:

`ERROR: meeting element at line` **X** `has` **Y** `occurrences of` **type**

where **X** is the line number in the input at which the corresponding `tMEETING` token appears, **Y** is the number of occurrences of the corresponding attribute, and **type** is the problematic attribute.

For example, the erroneous meeting element given above should produce the following error messages:

```
ERROR: meeting element at line 2 has 2 occurrences of start
ERROR: meeting element at line 2 has 0 occurrences of day
```

SR4: **Constraints cannot refer to an undefined code or undefined crn (15 points)**

In a constraint element, an item can mention either a code or a crn. The code/crn referenced in an item must be defined in the CSML file.

For example, suppose that we are given the following CSML file:

```
<course code="CS305" name='Programming Languages' type="Lecture">
    <class section="0" instructor='Husnu Yenigun' crn=20258 capacity=60>
        <meeting day=R start=08:40 end=10:30>
    </class>
</course>
<constraint>
   <item code="CS305"/>
   <item crn=20257/>
</constraint>
<course code="CS301R" name="Algorithms-Recitation" type="Recitation">
```

```
    <class section="0" instructor='Husnu Yenigun' crn=20257 capacity=50>
        <meeting start=17:40 day=M end=18:40/>
    </class>
</course>
<constraint>
     <item code="CS306"/>
     <item crn=20257/>
</constraint>
<constraint>
  <item code="CS305"/>
  <item crn=20258/>
</constraint>
```

The first `constraint` element given between the two `course` elements is valid, since this constraint refers to a course with `code="CS305"` and a section with a `crn=20257` which are both defined in the same CSML file. Note that, `crn=20257` is actually defined after the constraint element refers to it, but this is okay.

On the other hand, the second and the third `constraint` elements given at the end of the file have errors. The second `constraint` element refers to `code="CS306"` and the third `constraint` element refers to `crn=20258`, whereas neither a course with `code="CS306"` nor a class with `crn=20258` is defined in this CSML file.

When a constraint refers to an undefined code/crn, you should produce an error message in the following format:

`ERROR: constraint at line` **X** `refers to an undefined code=....`

or

`ERROR: constraint at line` **X** `refers to an undefined crn=....`

where **X** is the line number at which the token `tCONSTRAINT` appears.

For example, for the erroneous constraint elements given above you should produce the following error messages:

```
ERROR: constraint at line 15 refers to an undefined code="CS306"

ERROR: constraint at line 19 refers to an undefined crn=20258
```

3 Translation

If the input file is syntactically and semantically correct, then the translator part of your program must translate the input CSML program into an equivalent CSML program as explained below:

1. **Pretty print (20 points)**

   The whitespace characters in programs are typically used for increasing the readability of the source code. These whitespace characters usually don't affect the syntax/semantics of the programs. There are well–established conventions for the use of whitespace characters (e.g. every statement appears on a separate line, the constructs enclosed in another construct are indented to reflect the enclosure, etc.). A "pretty printer" is a translator from a programming language into the same programming language which simply reads a source code and re–writes it using the appropriate indentations, and by adding/removing other white space characters. Such pretty printers exist as a standalone translator, or sometimes they are part of Integrated Development Environments (IDEs). For example, in Visual Studio, as you type your code, it automatically corrects indentations to make the code more readable.

   The first feature that we will implement in our translator is such a pretty printer. The rules for pretty printing CSML programs are given below:

   (a) Each line contains only one tag.

   (b) When there are nested elements, inner elements have one more indent. Each indent is tab character. For example in Figure 1 at line 13, item tag is indented with one tab character since it is inner element of constraint element.

   (c) Unnecessary white spaces should be removed.

   ```
   <constraint  ><item  code="CS305"/><  item  crn=2057/></   constraint>
   ```

   Each indent is given by a tab.

   The below output is the pretty printed version of above example input.

   ```
   <constraint>
       <item code="CS305"/>
       <item crn=2057/>
   </constraint>
   ```

2. **Reordering the element list (20 points)**

   In CSML programs, the elements can be nested. However when we consider the top level, we see that a CSML program is a list of elements and these top level elements can be either a *course* element or a *constraint* element.

   On the other hand there is no restriction on the order of elements at the top level. In other words, the first top level element can be *course* element, and it can be followed by a *constraint* element, after which we can have another *course* element (e.g. see Figure 2 below).

   The translator part of your program must print out the same CSML program read from the input, by following a very simple ordering rule. First the *course*

```
<course code="CS305" name='Programming Languages' type="Lecture">
    <class section="0" instructor='Husnu Yenigun' crn=20258 capacity=60>
         <meeting start=17:40 day=M end=18:40/>
    </class>
</course>
<constraint>
    <item code="CS305"/>
    <item crn=2057/>
</constraint>
<course code="CS301R" name="Algorithms-Recitation" type="Recitation">
    <class section="0" instructor='Husnu Yenigun' crn=20257 capacity=50>
        <meeting start=17:40 day=M end=18:40/>
    </class>
</course>
```

Figure 2: Another example CSML program

elements must be listed. Then the constraint elements must be printed. You can assume that the ordering is applied only to main element list, the outer most element list.

As an example, for the CSML program given in Figure 2, your translator must produce the CSML program given in Figure 3.

4 Output

The program must print all the outputs (i.e. error message and the translated program) to the console.

If there are errors (violations of the semantic rules explained in Section 2) in the given CSML program, you must detect all these errors and print out the corresponding error messages, and then the program must terminate. That is, no translation will take place for an erroneous CSML program.

However if there are no errors in the given CSML program, then you must print out the translated form of the CSML program as explained in Section 3.

5 How to Submit

You can use the bison/flex files provided to you while developing the semantic checker and the translator for CSML. It is also acceptable if you modify these files and write your bison and flex files by yourselves.

```
<course code="CS305" name='Programming Languages' type="Lecture">
   <class section="0" instructor='Husnu Yenigun' crn=20258 capacity=60>
      <meeting start=17:40 day=M end=18:40/>
   </class>
</course>
<course code="CS301R" name="Algorithms-Recitation" type="Recitation">
   <class section="0" instructor='Husnu Yenigun' crn=20257 capacity=50>
      <meeting start=17:40 day=M end=18:40/>
   </class>
</course>
<constraint>
   <item code="CS305"/>
   <item crn=2057/>
</constraint>
```

Figure 3: Ordered form of the CSML program of Figure 2

Submit your files named as `id-hw4.y`, `id-hw4.flx`, `id-hw4.h`, and `id-hw4.c` where id is your student ID. We will compile your files by using the following commands:

```
flex id-hw4.flx
bison -d id-hw4.y
gcc -o id-hw4 lex.yy.c id-hw4.tab.c id-hw4.c -lfl
```

So, make sure that these three commands are enough to produce the executable semantic checker and translator. If we assume that there is a text file named `test` having a CSML program, we will execute your semantic checker/translator by using the following command line:

```
id-hw4 < test
```

The output should be displayed on the screen.

6 Notes

- **Important:** SUCourse's clock may be off a couple of minutes. Take this into account to decide when to submit.
- No homework will be accepted if it is not submitted using SUCourse.
- You must write your files by yourself.
- Start working on the homework immediately.