Programming Assignment #2

CS 6353 Section 1, Spring 2016
Updated: Feb 29 @ 2pm

The requirements for your submission are listed in sections 1 (Program Specification) and 2 (non-functional requirements). *Follow the requirements carefully and precisely – they will be thoroughly tested.*

Section 3 provides a few example outputs. Section 4 provides basic information on accessing and using the reference environment.

I grade by running your program against a large test suite, which includes many examples that are valid, as well as some samples that are invalid. You will be graded by the percentage of test cases you get correct.

NOTE that in Assignment 1 I curved things a bit by starting everybody at a base grade. Now that you've seen how I grade, I shouldn't have to do that—you're just expected to be thorough.

Therefore, I recommend you test thoroughly against good output to make sure your basic functionality is right; also test thoroughly against malformed output.

1. Program Specification

1. Your program must read 8-bit ASCII strings from standard input -- for instance, using the cin object in C++, or stdin in C. You must consume all input from standard input.
2. You are expected to tokenize the input stream using the input specification from Assignment 1, except that:
    a. You are no longer going to be outputting tokens (you'll use that token stream as input to your parser).
    b. You need to recognize two new token types:

Lexical Pattern	Numeric Type	English Type	Value to output
'('	33	(	None
')'	34	)	None

3. You then are expected to use the token stream and the BNF specification on the next page to parse the input into a parse tree.

Grammar BNF Specification:

```
<stmt>             ::= <while_stmt> <stmt>
                    |  <for_stmt> <stmt>
                    |  <if_stmt> <stmt>
                    |  <assignment_stmt> ';' <stmt>
                    |  <output_stmt> <stmt>
                    | ε
<while_stmt>       ::= 'while' '(' <cond_expr> ')' '{' <stmt> '}'
<for_stmt>         ::= 'for' '(' <opt_assign> ';' <opt_cond_expr>
                        ';' <opt_assign> ')' '{' <stmt> '}'
<if_stmt>          ::= 'if' '(' <cond_expr> ')' '{' <stmt> '}' <opt_else>
<opt_else>         ::= 'else' '{' <stmt> '}'
                    |  ε
<assignment_stmt> ::= ID ':=' <expr>
<output_stmt>      ::= '[' <expr> ']' ';'
<expr>             ::= <expr> '+' <mult_expr>
                    |  <mult_expr>
<mult_expr>        ::= <mult_expr> '*' <unary_expr>
                    |  <unary_expr>
<unary_expr>       ::= '!' <expr>
                    |  '+' <expr>
                    |  '-' <expr>
                    |  <paren_expr>
<paren_expr>       ::= '(' <expr> ')'
                    |  <basic_expr>
<basic_expr>       ::= ID
                    |  STR
                    |  INT
                    |  FLOAT
<cond_expr>        ::= <cond_expr> '|' <and_expr>
                    |  <and_expr>
<and_expr>         ::= <and_expr>  '&' <cmp_expr>
                    |  <eq_expr>
<eq_expr>          ::= <eq_expr>  '=' <cmp_expr>
                    |  <eq_expr> '!=' <cmp_expr>
                    |  <cmp_expr>
<cmp_expr>         ::= <cmp_expr> '<' <expr>
                    |  <cmp_expr> '>' <expr>
                    |  <cmp_expr> '<=' <expr>
                    |  <cmp_expr> '>=' <expr>
                    |  <expr>
<opt_cond_expr>    ::= <cond_expr>
                    |
<opt_assign>         ::= <assign_stmt>
                    | ε
```

4. Note the following about the above specification:
    a. Literals are put in single quotes, and all match back to the tokens from Assignment 1.
    b. Non-terminals are in angle-brackets.
    c. The character ε signifies the "empty string", meaning the rule matches on no input.
    d. IF there are any ambiguities in the grammar where multiple productions could be valid for a parse, you are expected to parse using the top-most production.
    e. The grammar above cannot be directly parsed using recursive descent. I recommend you try to refactor the grammar in places where recursive descent wouldn't work (per our discussion in class).
5. If the input string would have (in Assignment 1) generated an ERR* token, then your program should output the string: "`Lex error`" (without the quotes), followed by a single newline, and then it should exit with error code 0 (success). This string is case sensitive, and must be output to standard output.
6. If the token stream contains COMMENT tokens, NEWLINE tokens or WS tokens, ignore those tokens. They are not errors, and should never generate any output if they exist in a valid input.
7. If the token stream is valid per Assignment 1 (i.e., no ERR tokens would have been generated), but the resulting token stream does have tokens that are NOT USED by the grammar above, then your program should output the string: "`Unimplemented error`" (without the quotes), followed by a single newline, and then it should exit with error code 0 (success). This string is case sensitive, and must be output to standard output.
8. If the input string is not a string in the language represented by the grammar above (i.e.., if parsing fails), then you should output the string: "`Parse error`" (without the quotes), followed by a single newline, and then it should exit with error code 0 (success). This string is case sensitive, and must be output to standard output.
9. All specified output is CASE SENSITIVE.
10. Assuming the input is valid, your program must output a parse tree. The parse trees you output must have the following properties:
    a. Each node must either represent either a terminal or a non-terminal in the grammar above (or an equivalent transformation of the above grammar – you may make any transformation of the grammar for your own purposes that does not change the language matched – you can rename non-terminals and you can add productions that do not change the semantics—for instance, if needed to make the language parsable via recursive descent).
    b. Terminals must be either your tokens or the empty string (ε). Note that if you do not need the empty string in your graphs, that is fine!
    c. Terminal nodes must not have any children in the graph.
    d. There must be a single root node, and all nodes in the parse must be reachable from that root node.

   e. The graph must have left-to-right ordering—when you output the graph and I traverse it from left to right, I must be able to find all the tokens used in the parse IN ORDER.
   f. Your tree must be an unambiguous representation of the correct parse of the program. For instance, if you are parsing the expression: 1 + 2 * 3, it must be possible to determine that your graph properly represents the rules of precedence that are encoded in the grammar above.

11. Graphs will be output in a subset of XML described below. White space (spaces, horizontal tabs and newlines) in your output is unimportant between XML tokens, unless explicitly mentioned below.
12. All XML objects will be represented in the form: `<element_name>Value</element_name>`
    a. The element name, and must match the regular expression: `[_a-zA-Z][_a-zA-Z0-9]*`
    b. The "close tag" (e.g., `</entity_name>` must always be present—there is no "implicit close" allowed in your output.
13. When asked to output a terminal node representing ε, output: `<node><epsilon></epsilon></node>`
14. When asked to output a terminal node representing a token from the input, then output a node entity in the form: "`<node>%s</node>`", where you should substitute `%s` with the following required elements (which may appear in any order):
    a. *`id`*, which must contain the Token ID for the associated token, as set per Assignment 1. E.g., the first token (assuming it is not a whitespace token), should output: `<id>1</id>`
    b. *`typenum`*. The entity's contents should be the integer representing the token type from the chart in Assignment 1 (or in the supplemental table above).
    c. *`typename`*. The contents should be the type name associated with the integer token type.
    d. *position*. The contents should be the integer representing the offset of the start of the token into the original input string, as per Assignment 1.
    e. *`length`*. The contents should be the integer representing the length of the token, as per assignment 1.
    f. *`value`*. The contents should be the same as the associated output in Assignment 1. If there would have been no value output in Assignment 1, then this element is optional in your output. But if you do provide it, the contents must be empty.
15. For the above elements, they may exist no more than once in a single terminal node.
16. Additional elements may exist if you desire, but will be ignored.
17. For the above elements, all must exist in every token node, except as otherwise specified in 14f.
18. When outputting the value element for a token of type STR, you must not add additional white space into the element. For instance, if the string is "foo", your value element must be exactly: `<value>foo</value>`, and may not be: `<value>foo </value>`. This helps ensure that I can accurately recreate the original string contents.

19. When asked to output a non-terminal node, then output a node entity of the form: "<node>%s</node>", where you should substitute %s with the following required elements (which may appear in any order):
    a. `nt`, which must contain a string representing the name of your non-terminal (you can use names from the provided grammar, or change them if necessary).
    b. `children`, which must contain the full representation of all children nodes in the tree, IN THEIR LEFT TO RIGHT ORDER.
20. When your program finishes parsing, using the above rules, print the *root node* of your parse tree to stdout. Since the above specification is recursive, you will actually end up printing the entire parse tree.
21. When you have output your entire parse tree, your program must exit with error code 0 (success).
22. Command line arguments to your program shall be IGNORED.

2. Other Requirements

You will receive a 0 on this if any of these requirements are not met!

23. The assignment is due on March 14 at 8am Eastern time. Late assignments will lose one letter grade per 24 hours.
24. The program must be written entirely in C or C++
25. You must submit a single source code file, unless you choose to use multiple files, in which case you must submit a single ZIP file, and nothing else.
26. If submitting a ZIP file, when the file unzips, your source files must unzip into the same directory (including any header files you need).
27. If submitting a ZIP file, there must not be ANY other files contained within the ZIP file. Again, you will get a 0 if there are.
28. If your program is written in C, it must compile ON MY REFERENCE ENVIRONMENT into an executable with the following command line: `cc *.c –o assignment2`
29. If your program is written in C++, it must compile ON MY REFERENCE ENVIRONMENT into an executable with the following command line (**note this is slightly different than last time**): `c++ -std=c++11 *.cpp –o assignment2`
30. ***You may not use any 3rd party code for parser generation. I will inspect assignments to see if they are generated by any common tools.***
31. Your program should print nothing to stderr under any circumstances.
32. Your program's output will be tested in the reference environment only. Even if it works on your desktop, if it doesn't work in the reference environment, you will get a 0. With C and C++ this is a common occurrence due to memory errors, so be sure to test in the reference environment!
33. You must submit the homework through the course website, unless otherwise pre-approved by the professor.
34. You may not give or receive any help from other people on this assignment.
35. You may NOT use code from any other program, no matter who authored it.

3. Test Cases

Below are five sample test cases for you, which I will use in my testing. Note that, because you will need to transform the input grammar, the trees that you output will probably look different to the ones below. However, they must still correctly capture the syntax of the input string in tree form.

I will definitely use the below cases. I strongly recommend you create your own test harness and come up with a large number of test cases to help you get the best possible grade.

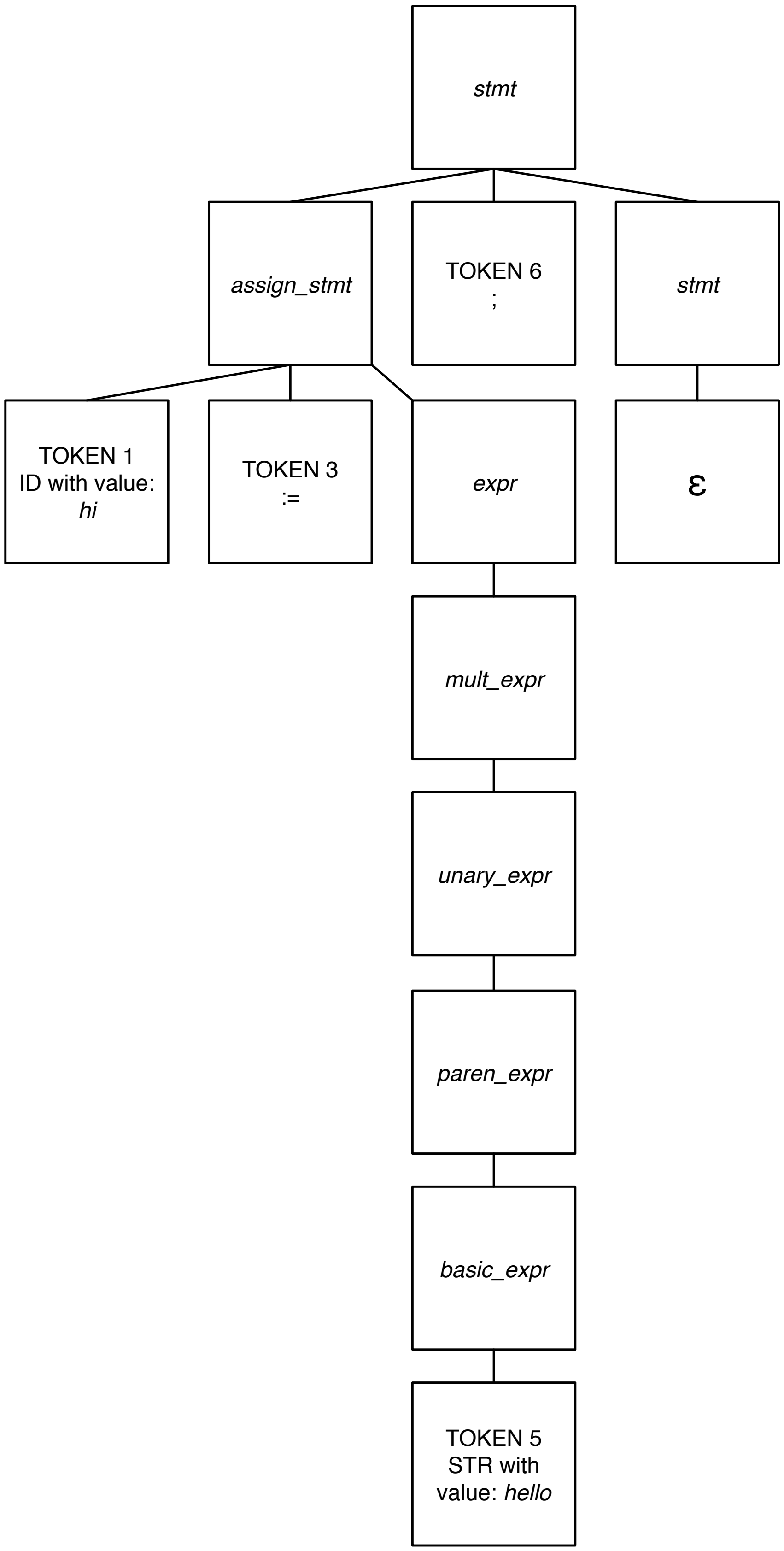
For test cases, what one would type on the command line is **BLACK**, input is in **GREEN**, and output is in **BLUE**.

Note that all the below test cases end with a final newline, and have no spaces before or after each line of input.

Case 1

```
./assignment2
hi := "hello";  # Assignment. This comment is part of the case.
<node><nt>stmt</nt><children>
  <node><nt>assign_stmt</nt><children>
    <node><id>1</id><typenum>1</typenum><typename>ID</typename>
        <position>0</position><length>2</length><value>hi</value>
    </node>
    <node><id>3</id><typenum>32</typenum><typename>:=</typename>
          <position>3</position><length>2</length>
    </node>
    <node><nt>expr</nt><children>
      <node><nt>mult_expr</nt><children>
        <node><nt>unary_expr</nt><children>
          <node><nt>paren_expr</nt><children>
            <node><nt>basic_expr</nt><children>
              <node><id>5</id><typenum>2</typenum><typename>STR</typename>
                    <position>6</position><length>7</length><value>hello</value>
              </node></children>
            </node></children>
          </node></children>
        </node></children>
      </node></children>
    </node></children>
  </node>
  <node>
      <id>6</id>
      <typenum>24</typenum>
      <typename>;</typename>
      <position>13</position>
      <length>1</length>
  </node>
  <node>
    <nt>stmt</nt><children>
      <node><epsilon></epsilon>
      </node></children>
  </node></children>
</node>
```

In graph form (with the token contents simplified), this would be:


stmt
assign_stmt
TOKEN 6 ;
stmt
TOKEN 1 ID with value: *hi*
TOKEN 3 :=
expr
ε
mult_expr
unary_expr
paren_expr
basic_expr
TOKEN 5 STR with value: *hello*

Case 2

```
./assignment2
"foo" "bar
12  # The unterminated string is an error per the lex spec.
Lex error
```

Case 3

```
./assignment2
135 % 23  # % doesn’t appear in the grammar, but is a valid token.
Unimplemented error
```

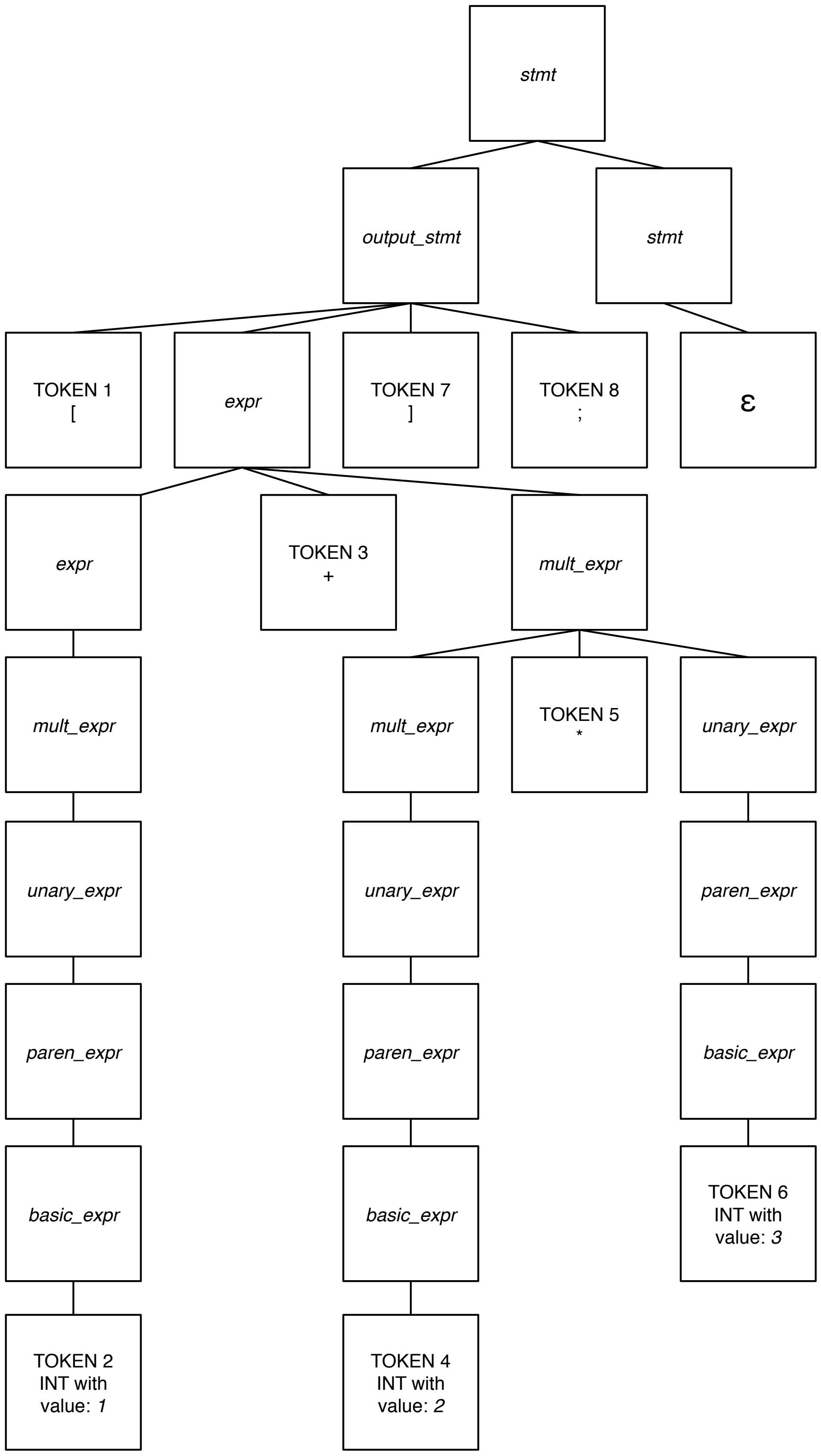
Case 4

```
./assignment2
hi;      # An error because ID; isn't valid in the language.
Parse error
```

Case 5

```
./assignment2 foo bar
[1+2*3]; # Note the extra command line args are ignored.
<node><nt>stmt</nt><children>
  <node><nt>output_stmt</nt><children>
    <node><id>1</id><typenum>22</typenum>
          <typename>[</typename><position>0</position>
          <length>1</length>
    </node>
    <node><nt>expr</nt><children>
      <node><nt>expr</nt><children>
        <node><nt>mult_expr</nt><children>
          <node><nt>unary_expr</nt><children>
            <node><nt>paren_expr</nt><children>
              <node><nt>basic_expr</nt><children>
                <node><id>2</id><typenum>3</typenum>
             <typename>INT</typename><position>1</position>
                  <length>1</length><value>1</value>
                </node></children>
              </node></children>
            </node></children>
          </node></children>
        </node></children>
      </node>
      <node><id>3</id><typenum>16</typenum><typename>+</typename>
            <position>2</position><length>1</length>
      </node>
      <node><nt>mult_expr</nt><children>
        <node><nt>mult_expr</nt><children>
          <node><nt>unary_expr</nt><children>
            <node><nt>paren_expr</nt><children>
              <node><nt>basic_expr</nt><children>
                <node><id>4</id><typenum>3</typenum>
                      <typename>INT</typename><position>3</position>
                      <length>1</length><value>2</value>
                </node></children>
              </node></children>
            </node></children>
          </node></children>
     </node>
        <node><id>5</id><typenum>18</typenum><typename>*</typename>
              <position>4</position><length>1</length>
        </node>
        <node><nt>unary_expr</nt><children>
          <node><nt>paren_expr</nt><children>
            <node><nt>basic_expr</nt><children>
              <node><id>6</id><typenum>3</typenum>
                    <typename>INT</typename><position>5</position>
                    <length>1</length><value>3</value>
              </node></children>
            </node></children>
          </node></children>
        </node></children>
      </node></children>
    </node>
    <node><id>7</id><typenum>23</typenum><typename>]</typename>
          <position>6</position><length>1</length>
    </node>
    <node><id>8</id><typenum>24</typenum><typename>;</typename>
          <position>7</position><length>1</length>
    </node>
</children>
</node>
  <node><nt>stmt</nt><children>
    <node><epsilon></epsilon>
    </node></children>
  </node></children>
</node>
```

In graph form (with the token contents simplified), this would be (see next page):


stmt
output_stmt
stmt
TOKEN 1 [
expr
TOKEN 7]
TOKEN 8 ;
ε
expr
TOKEN 3 +
mult_expr
mult_expr
mult_expr
TOKEN 5 *
unary_expr
unary_expr
unary_expr
paren_expr
paren_expr
paren_expr
basic_expr
basic_expr
basic_expr
TOKEN 6 INT with value: 3
TOKEN 2 INT with value: 1
TOKEN 4 INT with value: 2

4. Reference Environment

The reference environment remains kimota.net. Your credentials still work.