

Homework 2

Due: 14:00 pm EST, February 17, 2022 on Sakai

1. Consider the following attribute grammar for constant declaration:

1. Syntax rule: $\langle \text{const-declaration} \rangle \rightarrow \langle \text{type} \rangle \langle \text{id} \rangle = \langle \text{expr} \rangle$
Semantic rules: $\langle \text{id} \rangle.\text{type} \leftarrow \langle \text{type} \rangle.\text{type}$
 $\langle \text{expr} \rangle.\text{type} \leftarrow \langle \text{id} \rangle.\text{type}$
 $\langle \text{id} \rangle.\text{value} \leftarrow \langle \text{expr} \rangle.\text{value}$
2. Syntax rule: $\langle \text{type} \rangle \rightarrow \text{binary}$
Semantic rule: $\langle \text{type} \rangle.\text{type} \leftarrow \text{binary}$
3. Syntax rule: $\langle \text{type} \rangle \rightarrow \text{tertiary}$
Semantic rule: $\langle \text{type} \rangle.\text{type} \leftarrow \text{tertiary}$
4. Syntax rule: $\langle \text{expr} \rangle[1] \rightarrow \langle \text{expr} \rangle[2] \langle \text{const} \rangle$
Semantic rules: $\langle \text{expr} \rangle[1].\text{value} \leftarrow \text{if } \langle \text{expr} \rangle[1].\text{type} = \text{binary}$
 $\text{then } \langle \text{expr} \rangle[2].\text{value} * 2 + \langle \text{const} \rangle.\text{value}$
 $\text{else } \langle \text{expr} \rangle[2].\text{value} * 3 + \langle \text{const} \rangle.\text{value}$
 $\langle \text{expr} \rangle[2].\text{type} \leftarrow \langle \text{expr} \rangle[1].\text{type}$
5. Syntax rule: $\langle \text{expr} \rangle \rightarrow \langle \text{const} \rangle$
Semantic rule: $\langle \text{expr} \rangle.\text{value} \leftarrow \langle \text{const} \rangle.\text{value}$
6. Syntax rule: $\langle \text{const} \rangle \rightarrow 2$
Semantic rule: $\langle \text{const} \rangle.\text{value} = 2$
7. Syntax rule: $\langle \text{const} \rangle \rightarrow 1$
Semantic rule: $\langle \text{const} \rangle.\text{value} = 1$
8. Syntax rule: $\langle \text{const} \rangle \rightarrow 0$
Semantic rule: $\langle \text{const} \rangle.\text{value} = 0$
9. Syntax rule: $\langle \text{id} \rangle \rightarrow A \mid B \mid C$

- a. For each semantic rule, does the rule define a synthesized, inherited or intrinsic attribute?
- b. Given the input string "tertiary B = 210012"
 - i) Draw the parse tree representation for it. Is this input ambiguous?
 - ii) Draw the attribute dependency tree that shows the associated attributes at each node of the parse tree and their dependency relationships with attributes at other nodes (just like the tree in slide 59 in the week 3 lecture).
 - iii) Show the order of attribute evaluation according to the attribute dependencies. (just like the order of evaluations in the same slide 59 in the week 3 lecture).
 - iv) Draw the fully decorated (attributed) tree with evaluated attribute values at all nodes. Note: do the above steps on a single tree (akin to slide 60, week 3 lecture).

2. Recall the semantic function we dissected in week 3 under the denotational semantics for mapping assignment statements to states pertaining to logical pretest loops (i.e, while loops):

$$M_1(\text{while } B \text{ do } L, s) \Delta= \begin{array}{l} \text{if } M_b(B, s) == \text{undef} \\ \quad \text{then error} \\ \text{else if } M_b(B, s) == \text{false} \\ \quad \text{then } s \\ \quad \text{else if } M_{sl}(L, s) == \text{error} \\ \quad \quad \text{then error} \\ \quad \text{else } M_1(\text{while } B \text{ do } L, M_{sl}(L, s)) \end{array}$$

I want you to

- i) Describe, in natural language (i.e., English), what this function is doing, line by line. What are M_b and M_{sl} ?
- ii) Formally write a definition for M_b and M_{sl} that make this function complete. That is:
 - $M_b = ?$
 - $M_{sl} = ?$
- iii) The current semantic function, M_1 , is intended to model logical pretest loops. Write variation of it, M_1' , intended for logical posttest loops – *do while statements*; e.g., constructs that are of the form:

```
do {
    L
} while (B);
```

3. Derive the weakest precondition for the sequence of assignment statements and their postconditions below:

a) $x = 3 * y + 1;$
 $y = x - 5$
 $\{y < 1\}$

b) $c = 3 * (1 * b + c);$
 $b = 2 * c - 1$
 $\{b > 4\}$

4. Implement a recursive descent-parser in C, caller parser.c, for the following grammar (in EBNF) where `<program>` is the starting non-terminal. You can assume that the input source program is contained in an external textfile (e.g., 'front.in'), whose path name can be read from command line or in the main program. Note that whenever a symbol is in apostrophe, such as '`}`', it implies it is not part of the EBNF metalanguage and is part of the syntax rule itself.

`<program> → <stmts>`

`<stmts> → <stmt> [; <stmts>]`

`<stmt> → <assign> | <if> | <for> | <while> |`

`<assign> → id = <expr>`

`<if> → if '(' <expr> ')' ('{' <stmts> '}' | <stmt>)`

`<for> → for '(' <assign> ; <expr> ; <assign> ')' ('{' <stmts> '}' | <stmt>)`

`<while> → while '(' <expr> ')' ('{' <stmts> '}' | <stmt>)`

`<expr> → <term> { (+|-) <term> }`

`<term> → <factor> { (*|/) <factor> }`

`<factor> → <const> | <id> | '(' <expr> ')'`

`<id> → <letter> { <letter> | <digit> }`

`<const> -> <digit> { <digit> }`

`<letter> -> a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z`

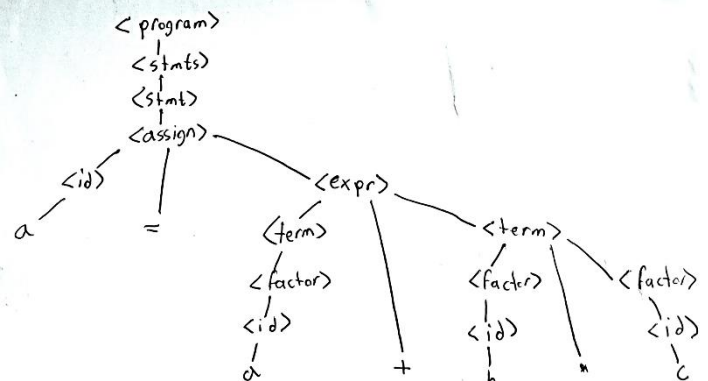
`<digit> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

Notice that the output perfectly models this tree:

Example Input: a = a + b * c

Output of the parser:

```
[<program> [<stmts> [<stmt>
[<assign> [<id>a] = [<expr> [
<term> [<factor> [<id>a]] +
[<term> [<factor> [<id>b]] *
[<factor> [<id>c]]]]]]]]]
```



Helpful Tips: a) Start by implementing (just means copy and make sure they work) the lexical and syntax analyzers in my lecture slides (i.e., textbook, chapters 4.2 to 4.3); this implies the copying of methods including but not limited to `lex()`, `expr()`, `term()`, where each of the syntax analyzer's methods like `expr()` and `term()` calls `lex()` to generate the parse.

b) Then, modifications/additions to the code that are necessary for this assignment specification include accounting for more syntax rules, such as those for non-terminals `<assign>`, `<for>`, `<stmts>`. This means the creation of new methods such as `assign()`, `for_statement()`, `stmts()` that will be similar in flavor to the ones you have in a). Note that what was initially a valid statement for the old syntax rules for arithmetic expressions is no longer valid in this new grammar (you can't have stand-alone expressions in the new grammar, such as `(sum+47)/total`).

Also, you will need to modify the code in the lecture slides/textbook in order for the output to be a nested representation of the tree and not just the output of the lexical analyzer + the steps to enter tree. That is, the output, if compared to a valid assignment statement with that expression (say, `a=(sum+47)/total`) would be different as follows:

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26 Next lexeme is )
Exit <factor>
Exit <term>
Next token is: 11 Next lexeme is total
Enter <factor>
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```



```
[<program>[<stmts>[<stmt>[<assign>
[<id>a]=[<expr>[<term>[<factor>([<e
xpr>[<term>[<factor>[<id>sum]]]+[<t
erm>[<factor>[<const>47]]]]]/[<facto
r>[<id>total]]]]]]]]]
```

(DON'T output this way)

DO output this way.

Finally, in the last page of this pdf, I have included more examples of input programs and their printed parse trees to help you test your program.

What to submit on Sakai:

1. A single pdf (e.g., assignment2.pdf) of your answers to all of the questions above (ideally typed, but you can also paste scans of handwritten answers on the pdf).
2. Your parser.c file.

```

a = a + b
[<program>[<stmts>[<stmt>[<assign>[<id>a]=[<expr>[<term>[<factor>[<id>a]]]+[<term>[<factor>[<id>b]]]]]]]]]]

a = a + b * c
[<program>[<stmts>[<stmt>[<assign>[<id>a]=[<expr>[<term>[<factor>[<id>a]]]+[<term>[<factor>[<id>b]]* [<factor>[<id>c]]]]]]]]]]

a = b * (c - d); e = f / g
[<program>[<stmts>[<stmt>[<assign>[<id>a]=[<expr>[<term>[<factor>[<id>b]]* [<factor>([<expr>[<term>[<factor>[<id>c]]]-[<term>[<factor>[<id>d]]]]]]]]]]];[<stmts>[<stmt>[<assign>[<id>e]=[<expr>[<term>[<factor>[<id>f]]/[<factor>[<id>g]]]]]]]]]]]]

if (a + b) c = d
[<program>[<stmts>[<stmt>[<if>if ([<expr>[<term>[<factor>[<id>a]]]+[<term>[<factor>[<id>b]]]]) [<stmt>[<assign>[<id>c]=[<expr>[<term>[<factor>[<id>d]]]]]]]]]]]]

if (a) {b = c/d; e=f-g}
[<program>[<stmts>[<stmt>[<if>if ([<expr>[<term>[<factor>[<id>a]]]]) {[<stmts>[<stmt>[<assign>[<id>b]=[<expr>[<term>[<factor>[<id>c]]]/[<factor>[<id>d]]]]]]];[<stmts>[<stmt>[<assign>[<id>e]=[<expr>[<term>[<factor>[<id>f]]]-[<term>[<factor>[<id>g]]]]]]]]]]]]]]]]

if (a) {b=c}
[<program>[<stmts>[<stmt>[<if>if ([<expr>[<term>[<factor>[<id>a]]]]) {[<stmts>[<stmt>[<assign>[<id>b]=[<expr>[<term>[<factor>[<id>c]]]]]]]]]]]]]]]]

while(a*b) c = d
[<program>[<stmts>[<stmt>[<while>
while ([<expr>[<term>[<factor>[<id>a]]* [<factor>[<id>b]]]]) [<stmt>[<assign>[<id>c]=[<expr>[<term>[<factor>[<id>d]]]]]]]]]]]]]]]]

while(a) {c = g + e; d = f*b}
[<program>[<stmts>[<stmt>[<while>
while ([<expr>[<term>[<factor>[<id>a]]]]) {[<stmts>[<stmt>[<assign>[<id>c]=[<expr>[<term>[<factor>[<id>g]]]+[<term>[<factor>[<id>e]]]]]]];[<stmts>[<stmt>[<assign>[<id>d]=[<expr>[<term>[<factor>[<id>f]]* [<factor>[<id>b]]]]]]]]]]]]]]]]

while(b) {a = f}
[<program>[<stmts>[<stmt>[<while>
while ([<expr>[<term>[<factor>[<id>b]]]]) {[<stmts>[<stmt>[<assign>[<id>a]=[<expr>[<term>[<factor>[<id>f]]]]]]]]]]]]]]]]

for(a=b;c;a=b+d) f = g
[<program>[<stmts>[<stmt>[<for>
for ([<assign>[<id>a]=[<expr>[<term>[<factor>[<id>b]]]]];[<expr>[<term>[<factor>[<id>c]]]]];[<assign>[<id>a]=[<expr>[<term>[<factor>[<id>b]]]+[<term>[<factor>[<id>d]]]]]) [<stmt>[<assign>[<id>f]=[<expr>[<term>[<factor>[<id>g]]]]]]]]]]]]]]

for(f=g;a*b;c=e-d) {a = b+c; d=g}
[<program>[<stmts>[<stmt>[<for>
for ([<assign>[<id>f]=[<expr>[<term>[<factor>[<id>g]]]]];[<expr>[<term>[<factor>[<id>a]]* [<factor>[<id>b]]]]];[<assign>[<id>c]=[<expr>[<term>[<factor>[<id>e]]]-[<term>[<factor>[<id>d]]]]]) {[<stmts>[<stmt>[<assign>[<id>a]=[<expr>[<term>[<factor>[<id>b]]]+[<term>[<factor>[<id>c]]]]]]];[<stmts>[<stmt>[<assign>[<id>d]=[<expr>[<term>[<factor>[<id>g]]]]]]]]]]]]]]]]

```