Milestone 2

Haiyang He

March 16, 2022

Required information

- name: Haiyang He
- UCID: 30067349
- email: haiyang.he@ucalgary.ca

- gitlab repo `https://gitlab.cpsc.ucalgary.ca/haiyang.he/cpsc411`

Briefly say what you're most proud of in this milestone's code and why I am proud of the following things:

- Good error messages with the line and column number! I found that it is possible to get the line number using bison (but not flex, at least not easily). This made debugging much easier and it feels like a pretty "smart" compiler!

  For example, you might write an invalid program like

  ```
  int f(){
      int a;
      a 3+3
  }
  ```

  The error message is

  ```
  testFiles/parserTests/testAround.txt: line 3, col 7: syntax error : expected ; before Number
  ```

  It is expecting a semi colon before a Number on line 3 column 7. And let's say you fixed it by doing an assignment:

  ```
  int f(){
      int a;
      a=3+3
  }
  ```

  The error message is

  ```
  testFiles/parserTests/testAround.txt: line 4, col 1: syntax error : expected ; before }
  ```

  It is expecting a semi colon before line 4 column 1, which is end of line 3. It will print out the line and column number when the parser fails, and uses bison specific functions to figure out what is expected before a token.

  This is better than the reference compiler as it only prints the line number.

- After discussing with my friend Jared, we noticed that if we have a chain of semi colons `;;;;`, the reference compiler will actually store them, which is a waste of space and hard to read the AST output. So I tried to get rid of the redundant semi colons inside the AST. We could have just traverse the AST after to get rid of it, but there should be a better way to not even store it in the first place. What I did is essentially searched where the "statement" rule can occur and inlined all of the possibilities with the semi colons.

  I realized that "blockstatement" can call "statement", but "blockstatements" call "blockstatement", so can inline the semi colon case in the "blockstatements" rule (by exahusting all possible cases with and without semi colon).

  Also, inside "statement", the 2 IF and 1 WHILE case calls "statement", so we can also exhaust all possibility there. Then the "statement" rule is not called in anymore places, so we can just remove the semi colon rule inside "statement".

  This actually works suprisingly well, since "statement" can only occur within a "blockstatement", which gets called from a "blockstatements", the grammar can accept things like `if(1);;;`, because this multiple semi colon case can ONLY occur within a block. Then since we added the explicit rules about semi colons, when we build the linked list inside the AST, each time it sees a semi colon it will just pass up a NULL pointer, so we actually just store 1 instead of a bunch of NULL pointers at the end.

- I also noticed that the reference compiler can parse statements like `if(1<2<3)`, which is technically not allowed by the grammar, since inside the IF condition it must be a statement expression, but `(1<2<3)` is not a statement expression. I think what happened is that Dr. Aycock was a bit lazy and made statement expression the same as expression. Thus here we will copy the reference compiler.

  But this does simplify things a lot, since to make the operator precedence and associativity correctly, bison can do that for you. With basic inspection of the grammar, we can easily get the precedence and associativity of the expression operators, specify them in bison and off we go. We don't need extra cases on the statement expression anymore.

Briefly say what in this milestone's code needs the most improvement and why One of the things that needs improvement might be my AST structure. I store things reasonably explicitly, since I just followed what the grammar structure looks like. But that made me writing a lot of different structs (and thus creating and printing them), which was a lot of work to do. To tag the types differently, I also had to make different enums for the structs that needed it. I think the number of different structs can be simplified more, but honestly I'm not too sure how to do that exactly, as in what can I simplify.

The parser also doesn't free the memory, but even though the assignment specification says we don't need to, it is still a limitation. The pretty printer for the AST is also not the greatest, since it prints out redundant new lines sometimes.

I also think I should make unit tests properly, rather than trying to run each of my test files (which are poorly named) everytime. But I simply don't have enough time to make it.

Identify one specific area in the code you'd like feedback on from the TA (it can be the area you think needs improvement, or a different area). Please identify the filename (and pathname, if applicable) of where that code can be found I would like to get some feedback on how to make the AST less messy, as in less than the number of cases I had to deal with. Is there some general rule I should follow for this? Also please provide me some feedback about my bison parser file `parser.y`, since this is my first time using bison and there can be things I'm doing that are not "standard practice".

What grade you think you've earned on the milestone and why We will give some evidence first:

- *Good error and warning messages output:* one example is already shown on the first page, but here are some more:

  This program is not valid in J– (we cannot declare and assign a variable at the same time):

```
int f(){
    int a=0;
}
```

parser outputs

```
testFiles/parserTests/testAround.txt: line 2, col 10: syntax error : expected ; before =
```

The scanner errors will also be printed with the following program:

```
int f(){
    int a;
    a = "hii \a
        newline";
}
```

has output

```
testFiles/parserTests/testAround.txt: line 3, col 15: warning: Illegal character escape in
side a string literal
testFiles/parserTests/testAround.txt: line 4, col 16: warning: Unexpected newline or carri
age return in string literal (missing terminating '")
testFiles/parserTests/testAround.txt: line 5, col 18: warning: Unexpected newline or carri
age return in string literal (missing terminating '")
testFiles/parserTests/testAround.txt: line 5, col 1: syntax error : expected ; before }
```

One thing that the reference compiler does it that it allows arguments in the main function, so programs like:

```
f(int a){
}
```

will be allowed by the reference compiler, but our parser outputs:

```
testFiles/parserTests/testAround.txt: line 1, col 3: syntax error : expected ) before int
```

which says we should close the parenthesis before typing int, which is what the grammar wants.

I think the line and column numbers are extremely helpful here too!

- *Predictable: correct inputs parsed correctly, erroneous inputs rejected.* we will give some test cases. The test files can be found at `./testFiles/parserTests/`.

  File `mytest1.txt`:

```
// just some basic testings
g(){
    if (1<2<3);
    (4<5)<6;
    boolean
    b
    ;
    int
    c;
```

```
    if (a==1)
        if (b==1)
            ;
        else
            d=2;
}

g(){
}
```

we output

```
mainDecl
        void, {'attr': None}
        Identifier, {'lineno': 2, 'colno': 1, 'attr': 'g'}
            formals:

        block:

            if {'lineno': 3, 'colno': 5, 'attr': None}
                < {'lineno': 3, 'colno': 12, 'attr': None}
                    < {'lineno': 3, 'colno': 10, 'attr': None}
                        Number, {'lineno': 3, 'colno': 9, 'attr': '1'}
                        Number, {'lineno': 3, 'colno': 11, 'attr': '2'}

                    Number, {'lineno': 3, 'colno': 13, 'attr': '3'}

                nullStatem

            statemExp: {'lineno': 4, 'attr': None}
                < {'lineno': 4, 'colno': 10, 'attr': None}
                    < {'lineno': 4, 'colno': 7, 'attr': None}
                        Number, {'lineno': 4, 'colno': 6, 'attr': '4'}
                        Number, {'lineno': 4, 'colno': 8, 'attr': '5'}

                    Number, {'lineno': 4, 'colno': 11, 'attr': '6'}


            varDecl:
                boolean, {'lineno': 5, 'colno': 5, 'attr': None}
                Identifier, {'lineno': 6, 'colno': 5, 'attr': 'b'}
            varDecl:
                int, {'lineno': 8, 'colno': 5, 'attr': None}
                Identifier, {'lineno': 9, 'colno': 5, 'attr': 'c'}
            if {'lineno': 11, 'colno': 5, 'attr': None}
                == {'lineno': 11, 'colno': 10, 'attr': None}
                    Identifier, {'lineno': 11, 'colno': 9, 'attr': 'a'}
                    Number, {'lineno': 11, 'colno': 12, 'attr': '1'}

                ifElse {'lineno': 12, 'colno': 9, 'attr': None}
                    == {'lineno': 12, 'colno': 14, 'attr': None}
                        Identifier, {'lineno': 12, 'colno': 13, 'attr': 'b'}
                        Number, {'lineno': 12, 'colno': 16, 'attr': '1'}

                    nullStatem

                    statemExp: {'lineno': 15, 'attr': None}
                        = {'lineno': 15, 'colno': 14, 'attr': None}
```

```
                            Identifier, {'lineno': 15, 'colno': 13, 'attr': 'd'}
                            Number, {'lineno': 15, 'colno': 15, 'attr': '2'}


mainDecl
        void, {'attr': None}
        Identifier, {'lineno': 18, 'colno': 1, 'attr': 'g'}
            formals:
```

The above test case shows the basic structure of AST, and the dangling else is handled "correctly" (since the else corresponds to the inside if).

File `mytest2.txt`:

```
        // more basic testings
int f(int b, boolean c){
    break;
    return a+1;
    return ;

    while(2)
    while(0)
    while(-1)
        2+3;
        int a;
    -2;
    !a;

    while(1)
        if (b)
            c=1;
            break;
    f(a,b,c);
    ------4;
    ------a;
}
```

outputs:

```
funcDecl
        int, {'lineno': 2, 'colno': 1, 'attr': None}
        Identifier, {'lineno': 2, 'colno': 5, 'attr': 'f'}
            formals:
                int, {'lineno': 2, 'colno': 7, 'attr': None}
                Identifier, {'lineno': 2, 'colno': 11, 'attr': 'b'}
                boolean, {'lineno': 2, 'colno': 14, 'attr': None}
                Identifier, {'lineno': 2, 'colno': 22, 'attr': 'c'}

        block:

            break: {'lineno': 3, 'colno': 5, 'attr': None}
            return: {'lineno': 4, 'colno': 5, 'attr': None}
                + {'lineno': 4, 'colno': 13, 'attr': None}
                    Identifier, {'lineno': 4, 'colno': 12, 'attr': 'a'}
                    Number, {'lineno': 4, 'colno': 14, 'attr': '1'}
```

```
return: {'lineno': 5, 'colno': 5, 'attr': None}

while: {'lineno': 7, 'colno': 5, 'attr': None}
    Number, {'lineno': 7, 'colno': 11, 'attr': '2'}
    while: {'lineno': 8, 'colno': 5, 'attr': None}
        Number, {'lineno': 8, 'colno': 11, 'attr': '0'}
        while: {'lineno': 9, 'colno': 5, 'attr': None}
            - {'lineno': 9, 'colno': 11, 'attr': None}
                Number, {'lineno': 9, 'colno': 12, 'attr': '1'}

            statemExp: {'lineno': 10, 'attr': None}
                + {'lineno': 10, 'colno': 10, 'attr': None}
                    Number, {'lineno': 10, 'colno': 9, 'attr': '2'}
                    Number, {'lineno': 10, 'colno': 11, 'attr': '3'}


varDecl:
    int, {'lineno': 11, 'colno': 9, 'attr': None}
    Identifier, {'lineno': 11, 'colno': 13, 'attr': 'a'}
statemExp: {'lineno': 12, 'attr': None}
    - {'lineno': 12, 'colno': 5, 'attr': None}
        Number, {'lineno': 12, 'colno': 6, 'attr': '2'}


statemExp: {'lineno': 13, 'attr': None}
    ! {'lineno': 13, 'colno': 5, 'attr': None}
        Identifier, {'lineno': 13, 'colno': 6, 'attr': 'a'}


while: {'lineno': 15, 'colno': 5, 'attr': None}
    Number, {'lineno': 15, 'colno': 11, 'attr': '1'}
    if {'lineno': 16, 'colno': 9, 'attr': None}
        Identifier, {'lineno': 16, 'colno': 13, 'attr': 'b'}
        statemExp: {'lineno': 17, 'attr': None}
            = {'lineno': 17, 'colno': 14, 'attr': None}
                Identifier, {'lineno': 17, 'colno': 13, 'attr': 'c'}
                Number, {'lineno': 17, 'colno': 15, 'attr': '1'}


break: {'lineno': 18, 'colno': 13, 'attr': None}
statemExp: {'lineno': 19, 'attr': None}
    funcCall{'lineno': 19, 'attr': None}
        Identifier, {'lineno': 19, 'colno': 5, 'attr': 'f'}
        func call args:
            Identifier, {'lineno': 19, 'colno': 7, 'attr': 'a'}
            Identifier, {'lineno': 19, 'colno': 9, 'attr': 'b'}
            Identifier, {'lineno': 19, 'colno': 11, 'attr': 'c'}


statemExp: {'lineno': 20, 'attr': None}
    - {'lineno': 20, 'colno': 5, 'attr': None}
        - {'lineno': 20, 'colno': 6, 'attr': None}
            - {'lineno': 20, 'colno': 7, 'attr': None}
                - {'lineno': 20, 'colno': 8, 'attr': None}
                    - {'lineno': 20, 'colno': 9, 'attr': None}
                        - {'lineno': 20, 'colno': 10, 'attr': None}
                            Number, {'lineno': 20, 'colno': 11, 'attr': '4'}
```

```
            statemExp: {'lineno': 21, 'attr': None}
                - {'lineno': 21, 'colno': 5, 'attr': None}
                    - {'lineno': 21, 'colno': 6, 'attr': None}
                        - {'lineno': 21, 'colno': 7, 'attr': None}
                            - {'lineno': 21, 'colno': 8, 'attr': None}
                                - {'lineno': 21, 'colno': 9, 'attr': None}
                                    - {'lineno': 21, 'colno': 10, 'attr': None}
                                        Identifier, {'lineno': 21, 'colno': 11, 'attr': 'a'}
```

This shows the behaviour of `break`, `return`, `while`, and basic unary operators.
File `mytest3.txt`:

```
void f(int a, int b, boolean c){
    {
        int a;
        {
            int b;
        }
    }
    {
        int c;
    }
    return 2+(3+5);
}
```

output is

```
funcDecl
        void, {'lineno': 2, 'colno': 1, 'attr': None}
        Identifier, {'lineno': 2, 'colno': 6, 'attr': 'f'}
            formals:
                int, {'lineno': 2, 'colno': 8, 'attr': None}
                Identifier, {'lineno': 2, 'colno': 12, 'attr': 'a'}
                int, {'lineno': 2, 'colno': 15, 'attr': None}
                Identifier, {'lineno': 2, 'colno': 19, 'attr': 'b'}
                boolean, {'lineno': 2, 'colno': 22, 'attr': None}
                Identifier, {'lineno': 2, 'colno': 30, 'attr': 'c'}

        block:

            block:

                varDecl:
                    int, {'lineno': 4, 'colno': 9, 'attr': None}
                    Identifier, {'lineno': 4, 'colno': 13, 'attr': 'a'}
                block:

                    varDecl:
                        int, {'lineno': 6, 'colno': 13, 'attr': None}
                        Identifier, {'lineno': 6, 'colno': 17, 'attr': 'b'}
```

```
        block:

            varDecl:
                int, {'lineno': 10, 'colno': 9, 'attr': None}
                Identifier, {'lineno': 10, 'colno': 13, 'attr': 'c'}
        return: {'lineno': 12, 'colno': 5, 'attr': None}
            + {'lineno': 12, 'colno': 13, 'attr': None}
                Number, {'lineno': 12, 'colno': 12, 'attr': '2'}
                + {'lineno': 12, 'colno': 16, 'attr': None}
                    Number, {'lineno': 12, 'colno': 15, 'attr': '3'}
                    Number, {'lineno': 12, 'colno': 17, 'attr': '5'}
```

This shows it parses the nested blocks correctly.

File `mytest4.txt`:

```
//testing basic operator precedence
int g(){
    c+a<3;
    c != a == b;
    c <= a == d > b ;
    c + -a / d ;
    !c %d >= 1;
    1<3>d %11;
    2||23||3;
}
```

outputs

```
funcDecl
        int, {'lineno': 2, 'colno': 1, 'attr': None}
        Identifier, {'lineno': 2, 'colno': 5, 'attr': 'g'}
            formals:

        block:

            statemExp: {'lineno': 3, 'attr': None}
            < {'lineno': 3, 'colno': 8, 'attr': None}
                + {'lineno': 3, 'colno': 6, 'attr': None}
                    Identifier, {'lineno': 3, 'colno': 5, 'attr': 'c'}
                    Identifier, {'lineno': 3, 'colno': 7, 'attr': 'a'}

                Number, {'lineno': 3, 'colno': 9, 'attr': '3'}


            statemExp: {'lineno': 4, 'attr': None}
            == {'lineno': 4, 'colno': 12, 'attr': None}
                != {'lineno': 4, 'colno': 7, 'attr': None}
                    Identifier, {'lineno': 4, 'colno': 5, 'attr': 'c'}
                    Identifier, {'lineno': 4, 'colno': 10, 'attr': 'a'}

                Identifier, {'lineno': 4, 'colno': 15, 'attr': 'b'}


            statemExp: {'lineno': 5, 'attr': None}
            == {'lineno': 5, 'colno': 12, 'attr': None}
                <= {'lineno': 5, 'colno': 7, 'attr': None}
```

```
                Identifier, {'lineno': 5, 'colno': 5, 'attr': 'c'}
                Identifier, {'lineno': 5, 'colno': 10, 'attr': 'a'}

            > {'lineno': 5, 'colno': 17, 'attr': None}
                Identifier, {'lineno': 5, 'colno': 15, 'attr': 'd'}
                Identifier, {'lineno': 5, 'colno': 19, 'attr': 'b'}



statemExp: {'lineno': 6, 'attr': None}
    + {'lineno': 6, 'colno': 7, 'attr': None}
        Identifier, {'lineno': 6, 'colno': 5, 'attr': 'c'}
        / {'lineno': 6, 'colno': 12, 'attr': None}
            - {'lineno': 6, 'colno': 9, 'attr': None}
                Identifier, {'lineno': 6, 'colno': 10, 'attr': 'a'}

            Identifier, {'lineno': 6, 'colno': 14, 'attr': 'd'}



statemExp: {'lineno': 7, 'attr': None}
    >= {'lineno': 7, 'colno': 11, 'attr': None}
        % {'lineno': 7, 'colno': 8, 'attr': None}
            ! {'lineno': 7, 'colno': 5, 'attr': None}
                Identifier, {'lineno': 7, 'colno': 6, 'attr': 'c'}

            Identifier, {'lineno': 7, 'colno': 9, 'attr': 'd'}

        Number, {'lineno': 7, 'colno': 14, 'attr': '1'}


statemExp: {'lineno': 8, 'attr': None}
    > {'lineno': 8, 'colno': 8, 'attr': None}
        < {'lineno': 8, 'colno': 6, 'attr': None}
            Number, {'lineno': 8, 'colno': 5, 'attr': '1'}
            Number, {'lineno': 8, 'colno': 7, 'attr': '3'}

        % {'lineno': 8, 'colno': 11, 'attr': None}
            Identifier, {'lineno': 8, 'colno': 9, 'attr': 'd'}
            Number, {'lineno': 8, 'colno': 12, 'attr': '11'}



statemExp: {'lineno': 9, 'attr': None}
    || {'lineno': 9, 'colno': 10, 'attr': None}
        || {'lineno': 9, 'colno': 6, 'attr': None}
            Number, {'lineno': 9, 'colno': 5, 'attr': '2'}
            Number, {'lineno': 9, 'colno': 8, 'attr': '23'}

        Number, {'lineno': 9, 'colno': 12, 'attr': '3'}
```

This shows it can handle operator precedence.

File `mytest5.txt`:

```
f(){
    x = "123";
    x = true;
```

```
    x = 2;
}

g(){

    a = b = - b + c * 2 - (1+3);
    //can be parsed as
    // a = (b = (((-b) + (c * 2)) - (1+3)))


    return 3 -!-- a + b * c;
    //can be parsed as
    // 3 - ( (!(-(-a))) + (b * c));

}
```

outputs:

```
mainDecl
        void, {'attr': None}
        Identifier, {'lineno': 1, 'colno': 1, 'attr': 'f'}
            formals:

        block:

            statemExp: {'lineno': 2, 'attr': None}
                = {'lineno': 2, 'colno': 7, 'attr': None}
                    Identifier, {'lineno': 2, 'colno': 5, 'attr': 'x'}
                     , {'lineno': 2, 'colno': 13, 'attr': '"123"'}


            statemExp: {'lineno': 3, 'attr': None}
                = {'lineno': 3, 'colno': 7, 'attr': None}
                    Identifier, {'lineno': 3, 'colno': 5, 'attr': 'x'}
                    true, {'lineno': 3, 'colno': 9, 'attr': None}


            statemExp: {'lineno': 4, 'attr': None}
                = {'lineno': 4, 'colno': 7, 'attr': None}
                    Identifier, {'lineno': 4, 'colno': 5, 'attr': 'x'}
                    Number, {'lineno': 4, 'colno': 9, 'attr': '2'}


mainDecl
        void, {'attr': None}
        Identifier, {'lineno': 7, 'colno': 1, 'attr': 'g'}
            formals:

        block:

            statemExp: {'lineno': 9, 'attr': None}
                = {'lineno': 9, 'colno': 7, 'attr': None}
                    Identifier, {'lineno': 9, 'colno': 5, 'attr': 'a'}
                    = {'lineno': 9, 'colno': 11, 'attr': None}
                        Identifier, {'lineno': 9, 'colno': 9, 'attr': 'b'}
                        - {'lineno': 9, 'colno': 25, 'attr': None}
                            + {'lineno': 9, 'colno': 17, 'attr': None}
                                - {'lineno': 9, 'colno': 13, 'attr': None}
```

```
                                    Identifier, {'lineno': 9, 'colno': 15, 'attr': 'b'}

                                * {'lineno': 9, 'colno': 21, 'attr': None}
                                    Identifier, {'lineno': 9, 'colno': 19, 'attr': 'c'}
                                    Number, {'lineno': 9, 'colno': 23, 'attr': '2'}


                            + {'lineno': 9, 'colno': 29, 'attr': None}
                                Number, {'lineno': 9, 'colno': 28, 'attr': '1'}
                                Number, {'lineno': 9, 'colno': 30, 'attr': '3'}




        return: {'lineno': 14, 'colno': 5, 'attr': None}
            + {'lineno': 14, 'colno': 21, 'attr': None}
                - {'lineno': 14, 'colno': 14, 'attr': None}
                    Number, {'lineno': 14, 'colno': 12, 'attr': '3'}
                    ! {'lineno': 14, 'colno': 15, 'attr': None}
                        - {'lineno': 14, 'colno': 16, 'attr': None}
                            - {'lineno': 14, 'colno': 17, 'attr': None}
                                Identifier, {'lineno': 14, 'colno': 19, 'attr': 'a'}




                * {'lineno': 14, 'colno': 25, 'attr': None}
                    Identifier, {'lineno': 14, 'colno': 23, 'attr': 'b'}
                    Identifier, {'lineno': 14, 'colno': 27, 'attr': 'c'}
```

This shows it can parse complex expressions.

File `mytest6.txt`:

```
//empty if statement condition
void f(){
    if ();
}
```

outputs:

```
testFiles/parserTests/mytest6.txt: line 3, col 9: syntax error  before )
```

This shows it recognizes the if statement condition must be non-empty, hence syntax error at the right parenthesis.

File `mytest7.txt`:

```
//test empty/NULL AST structures
f(){
    //  ---------------------
    //  testing if statements
    //  ---------------------

    //if statement with empty body
    if (1);
```

```
    //if with non-empty body
    if (1){
        int a;
    }
    //if and else with both empty bodies
    if (2);
    else;

    //if with both empty body but else is non-empty body
    if (2);
    else{
        int b;
    };

    //if with non-empty body but else with empty body
    if (3){
        int c;
    }
    else;

    //if and else with non-empty body
    if (3){
        int d;
    }
    else{
        int e;
    }

    //  ----------------------
    //  testing function invocation
    //  ----------------------
    f();
    f(a,b);

}
```

outputs:

```
mainDecl
        void, {'attr': None}
        Identifier, {'lineno': 2, 'colno': 1, 'attr': 'f'}
            formals:

        block:

            if {'lineno': 8, 'colno': 5, 'attr': None}
                Number, {'lineno': 8, 'colno': 9, 'attr': '1'}
                nullStatem

            if {'lineno': 11, 'colno': 5, 'attr': None}
                Number, {'lineno': 11, 'colno': 9, 'attr': '1'}
                block:

                    varDecl:
                        int, {'lineno': 12, 'colno': 9, 'attr': None}
                        Identifier, {'lineno': 12, 'colno': 13, 'attr': 'a'}
            ifElse {'lineno': 15, 'colno': 5, 'attr': None}
```

```
        Number, {'lineno': 15, 'colno': 9, 'attr': '2'}
        nullStatem

        nullStatem

    ifElse {'lineno': 19, 'colno': 5, 'attr': None}
        Number, {'lineno': 19, 'colno': 9, 'attr': '2'}
        nullStatem

        block:

            varDecl:
                int, {'lineno': 21, 'colno': 9, 'attr': None}
                Identifier, {'lineno': 21, 'colno': 13, 'attr': 'b'}
    ifElse {'lineno': 25, 'colno': 5, 'attr': None}
        Number, {'lineno': 25, 'colno': 9, 'attr': '3'}
        block:

            varDecl:
                int, {'lineno': 26, 'colno': 9, 'attr': None}
                Identifier, {'lineno': 26, 'colno': 13, 'attr': 'c'}
        nullStatem

    ifElse {'lineno': 31, 'colno': 5, 'attr': None}
        Number, {'lineno': 31, 'colno': 9, 'attr': '3'}
        block:

            varDecl:
                int, {'lineno': 32, 'colno': 9, 'attr': None}
                Identifier, {'lineno': 32, 'colno': 13, 'attr': 'd'}
        block:

            varDecl:
                int, {'lineno': 35, 'colno': 9, 'attr': None}
                Identifier, {'lineno': 35, 'colno': 13, 'attr': 'e'}
    statemExp: {'lineno': 41, 'attr': None}
        funcCall{'lineno': 41, 'attr': None}
            Identifier, {'lineno': 41, 'colno': 5, 'attr': 'f'}
            func call args:


    statemExp: {'lineno': 42, 'attr': None}
        funcCall{'lineno': 42, 'attr': None}
            Identifier, {'lineno': 42, 'colno': 5, 'attr': 'f'}
            func call args:
                Identifier, {'lineno': 42, 'colno': 7, 'attr': 'a'}
                Identifier, {'lineno': 42, 'colno': 9, 'attr': 'b'}
```

This shows it can handle all types of if-else statements correctly, as it can recognize if there is only 1 IF, or there is a IF-ELSE.

File `mytest8.txt`:

```
//testing sequence of semi colons
void g(){ }
void h(int a){ }

f(){
```

```
    g(a,b,c);
    int a;;;;;;;
    if(1);;;;;

    if(1){}
    else;;;;;

    if(1){;;}
    else;;;;;{;;}
}
```

outputs:

```
funcDecl
        void, {'lineno': 2, 'colno': 1, 'attr': None}
        Identifier, {'lineno': 2, 'colno': 6, 'attr': 'g'}
            formals:


funcDecl
        void, {'lineno': 3, 'colno': 1, 'attr': None}
        Identifier, {'lineno': 3, 'colno': 6, 'attr': 'h'}
            formals:
                int, {'lineno': 3, 'colno': 8, 'attr': None}
                Identifier, {'lineno': 3, 'colno': 12, 'attr': 'a'}


mainDecl
        void, {'attr': None}
        Identifier, {'lineno': 5, 'colno': 1, 'attr': 'f'}
            formals:

        block:

            statemExp: {'lineno': 6, 'attr': None}
                funcCall{'lineno': 6, 'attr': None}
                    Identifier, {'lineno': 6, 'colno': 5, 'attr': 'g'}
                    func call args:
                        Identifier, {'lineno': 6, 'colno': 7, 'attr': 'a'}
                        Identifier, {'lineno': 6, 'colno': 9, 'attr': 'b'}
                        Identifier, {'lineno': 6, 'colno': 11, 'attr': 'c'}


            varDecl:
                int, {'lineno': 7, 'colno': 5, 'attr': None}
                Identifier, {'lineno': 7, 'colno': 9, 'attr': 'a'}
            if {'lineno': 8, 'colno': 5, 'attr': None}
                Number, {'lineno': 8, 'colno': 8, 'attr': '1'}
                nullStatem

            ifElse {'lineno': 10, 'colno': 5, 'attr': None}
                Number, {'lineno': 10, 'colno': 8, 'attr': '1'}
                block:

                nullStatem

            ifElse {'lineno': 13, 'colno': 5, 'attr': None}
                Number, {'lineno': 13, 'colno': 8, 'attr': '1'}
```

```
            block:

            nullStatem

        block:
```

This shows it doesn't store extra semi colons, while it parses the file correctly.

File `mytest9.txt`:

```
main(){
    f(b, int a, c);
}
```

output:

```
testFiles/parserTests/mytest9.txt: line 2, col 10: syntax error  before int
```

We cannot declare a variable when we call a function, and the parser catches this by providing the line and column number of the keyword `int`, which is very nice!

File `mytest10.txt`:

```
f(){
    boolean a
    what c;
}
```

outputs:

```
testFiles/parserTests/mytest10.txt: line 3, col 5: syntax error : expected ; before Identifier
```

Again, since it tells you the column number, on line 3 column 5, it is the word `what`. It thinks `what` is an variable/identifier, and it notices there is no semi colon after `a`.

Not sure how much the above test cases can show/convince, but hopefully it is somewhat convincing that the parser works.

- *Committing work regularly to your CPSC Gitlab repo:* I have a reasonable amount of commits, and I push when I get a big chunk done so I don't lose the files accidentally!
- *Code is modular:* This time I had special functions (logging functions) for error messages, which was nice. The AST structures are contained in `./ast.h` and `./ast.c`, so the `./parser.y` only needs to know a few types to set up the return values of the bison rules. For each AST structure there is a corresponding "create" function, so we can just call that in bison so that was quite convenient (although not convenient to write the corresponding create and printing functions).
- *Code is extensible:* it is not very extensible, in the sense that if the grammar is changed drastically, many parts of the AST will need to be rewritten. This is because I created the different structures to essentially match the grammar, which means the AST is closely tied to the grammar rules. However, it is not hard to add in small things, since I used unions and enums to tag and distinguish the stored types. If we want to store something more in the grammar, we can just add it to the union.
- *Code is consistent e.g., with respect to variable and function naming conventions:* I think my naming conventions are mostly consistent. For example, the global variables and enum members are all in capital letters with underscores. The structure and variable names are camel cased. If you want to create or print something, the corresponding functions have the prefix "create" or "print", and the type name is appended at the back. For example, the functions corresponding to the structure `Expression` are `createExpression` and `printExpression`.

- *Code is consistent with respect to the implementation language, i.e., are appropriate idioms and libraries used?:* I think the code is consistent with the implementation language. I used enums and unions in C, which creates a reasonably type safe sum type. I used bison to handle the operator precedence and associativity. I also had to change my scanner to set the `yylval` to a structure that contains the line and column number, and possibly a string, for the operators and reserved keywords so I can store them in my AST using bison. Also, since bison can generate the enum tags for operators and reserved words, I modified my scanner to use the bison generated enum instead of my own enum, so that bison and flex can communicate with each other correctly.

- *Code is well documented:* I provided some documentation to the `parser.y` file to explain what I changed around in the parser. I also commented in `ast.h`, which explains what each structure is and what they store. However, since the structures are mostly self explanatory, the comments were not given in huge detail. But it does express how the AST is structured and what it stores clearly.

- *Speed: code should not be unreasonably slow* I used bison for generating the AST, so I don't think that will be too slow, since all I do in bison is malloc the struct and chain them together. As for printing the AST, I basically did a DFS through the nodes and printed them. The indentation is reprintted every time, but the alternative is to pass down a C string, which require much more work. Thus I don't think my program is very slow.

  Actually, I had to reverse the linked lists (such as the top level declarations and function arguments) as I return them from the AST. I could not reverse it if I'm doing right recursion, which is something like

  ```
  rule    : ruleLeaf
          | ruleLeaf rule
  ```

  Then that way I can just attach the new `ruleLeaf` as the head. However, this will introduce right recursion and since bison is LALR parser, right recursion is not ideal since it will push a lot of stack space. Thus I stayed left recursively and reversed the linked list at the very end (not at each step sine that will be $O(n^2)$). So for each linked list in the AST, an extra $O(n)$ factor is required to built it, which I think it is reasonable.

- *Easy to build, and building without errors or warnings:* Here is the following build output on the CPSC machines:

  ```
  haiyang.he@csx3:~/cpsc411$ make parser
  bison -v -Wall -d parser.y
  parser.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
  parser.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
  flex lexer.l
  gcc -Wall parserMain.c scanner.o commonHeader.o parser.tab.o ast.o -lfl -o parser.out
  ```

  so it builds fine with 1 shift reduce conflict. If we compile with `-Wcounterexamples` we can see the conflict is the dangling else. I cannot show it here since latex doesn't compile with a unicode symbol inside the verbatim environment.

- *Tool usage, if applicable:* as I said above, I used bison and flex. I had to change up my scanner code to work with bison (modified the return type and `yylval`), so the pattern-matching facilities of the scanner generator should be used appropriately.

- *Consistent with the environment: error and warning messages sent to stderr, regular output sent to stdout, exit status code set appropriately:* I print the warning and error messages with stderr, and otherwise we print to stdout. I also set the exit code correctly in `./parserMain.c` by returning `exit(EXIT_FAILURE)` if either the parser or the scanner had an error, and `return 0` otherwise, which indicates a successful execution.

- *Easy to run:* I only allowed 1 input argument for my parser, so I think it is easy to run.

- *Meta-documentation: is the way to build and run your milestone code clearly documented in the repo's README.md ?* Yes you can find the `README.md` which explains how to build and run the milestone.

Finally, I think I did what the milestone should do. I took it a little further by specifying column numbers and thus better error messages, and I was careful of not doing right recursion in bison. I also observed the behaviour of the reference compiler and changed the parser accordingly (not storing extra semi colons and making statement expression the same as expression). Thus overall I think I should get somewhere between 7/8 and 8/8.