CSE340F22 PROJECT 2

IMPLEMENTATION GUIDANCE

Rida Bazzi

This is not meant to be a detailed implementation guide, but I address major implementation issues that you are likely to encounter.

By now you should know that reading everything carefully is essential and can save you a lot of time.

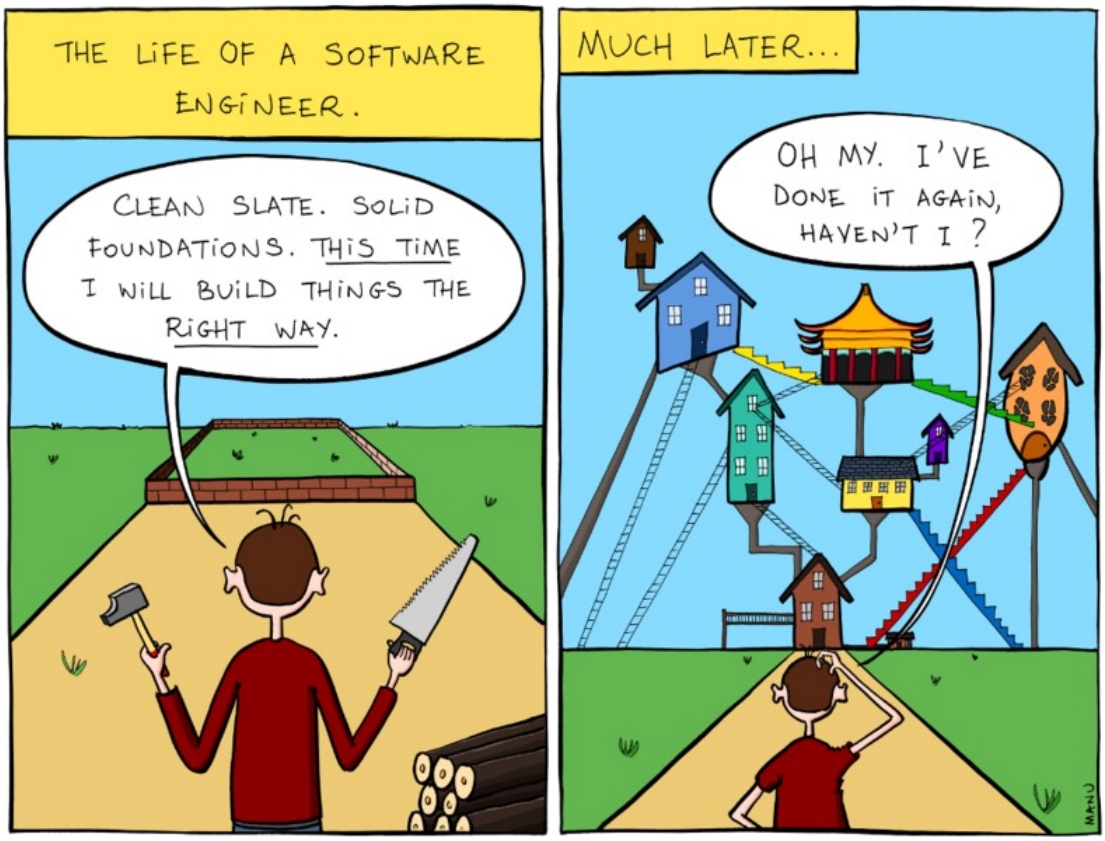
You should have a plan and understand how the various pieces will fit together before you start coding.

You can do Task 1 without worrying how task 2 or 3 will be implemented.

You can do Task 2 without worrying how Task 3 will be implemented because Task 3 will build on the abstract syntax tree generated by Task 1 and can you that as a starting point.

Do not delay asking for help.

If your implementation seems to be getting too complicated, especially conceptually, you should step back and simplify. Do not end up like this guy:



Parsing Concerns

Operator precedence parsing is at the heart of all the tasks for this project, so you really cannot afford not doing it right. It is very important that you have a complete understanding of how operator precedence parsing works before writing a single line of code.

In project 1 some students tried recursive descent parsing by trial-and-error instead of systematically following the rules for predictive parsing that I covered in class. That did not always go well and in the best case, is too time consuming. Parsing by trial-and-error will not work for this project and you really need to know what you are doing. As a start, you should have studied the operator precedence parsing notes before reading this document.

For this project, operator precedence parsing will be invoked for parsing expressions. There are four places where that occurs:

```
ID [.]     = expr;
ID         = expr;
ID [expr]  = expr;
OUTPUT ID[expr];
```

So, the function to parsing assignment statements might look something like this (this code is not meant to be copied and pasted. It is only for illustration purposes):

```
parse_assign_stmt()
{
    expect(ID)
    t1 = lexer.peek(1);
    t2 = lexer.peek(2);
    if ( t1t2 == [. )        // array access with .
    {
        expect(LBRAC);
        expect(DOT);
        expect(RBRAC);
    } else if (t1 == [ )     // array access but not .
    {
        expect(LBRAC);
        parse_expr();
        expect(RBRAC);
    }
    // at this point we have parsed the [.] or [expr]
    // or we are dealing with the case ID =
    // in all cases, the next token must be EQUAL
    expect(EQUAL);
    parse_expr();
    expect(SEMICOLON);
}
```

As you can see, the recursive-descent part of the parsing is really not that involved. OUTPUT statements can also be handled in a similar manner. It is the operator precedence parsing where the bulk of your work is done.

Next we are going to examine the call to `parse_expr()` more carefully.

`parse_expr()`: what is the end-of-input symbol?

As we have seen in the notes on operator precedence parsing, we start with a stack containing $ and the input is followed by the same $ symbol.

In our project, `parse_expr()` is called in three settings. In one setting, the part of the input to be parsed as an expression is followed by SEMICOLON. In two other settings, the input is followed by RBRAC. They are highlighted below:

```
ID[.]          = expr  ;     (1)
ID             = expr  ;     (2)
ID[ expr  ] = expr  ;     (3)   (4)
OUTPUT ID[ expr ] ;          (5)
```

To handle these different situations, the `parse_expr()` needs to identify when the end of input is reached for the call. The cases outlined above can be problematic in the case of RBRAC in variable-access or OUTPUT statement because it can also appear as part of *expr* itself. So, instead of RBRAC, we can use RBRAC EQUAL to indicate end of input in the case of variable access and RBRAC SEMICOLON in the case of OUTPUT statement

To handle these cases, I suggest the following:

1. Instead of using `GetToken()` and `peek()` directly inside `parse_expr()`, wrap the call inside functions `get_symbol()` and `peek_symbol()` respectively. The functions `peek_symbol()` returns $ (or `EOE` = End Of Expression as one student suggested to me after class) in the following cases:
    1. If the next token is SEMICOLON ; cases (1) , (2) and (4)
    2. If the next token is RBRAC and the token after that is EQUAL ; case (3)
    3. If the next token is RBRAC and the token after that is SEMICOLON ; case (5)
2. The functions return the next token in all other cases.

Having a wrapper function will make your code easier to work with

parse_expr(): parsing table and stack contents

As we have seen in class, the parsing table is convenient to check the parsing precedence relationship for two terminals (this includes $).

1. We need the table to determine the relationship between the terminal closest to the top of the stack and the next input symbol.
2. We also need the table to determine the relationship between the terminal on the top of the stack and the last popped terminal when we are popping the stack to identify the RHS of a rule.

We start by discussing the format of the stack

both terminals and non-terminals (expr) are stored on the stack, so the stack entries should make the distinction between these two cases. For that, you can can have a structure

```
struct stackNode {
   snodeType type;          // enum type can be EXPR or TERM
   union {
      struct exprNode *expr;  // this is used when the type is EXPR
      Token term;             // this is used when the type is TERM
   }
}
```

The `union` declaration is used like a structure when only one field is needed for a particular instance. It allows to use the same memory locations for the two fields, which results in less memory usage. I also like to use `union` instead of struct because by declaring the fields as fields of a union, you are documenting the fact that for a given `stackNode`, you are only either using the `expr` field or the `term` field. The type field tells you which case it is. Now, the stack will contain `stackNode` structs that can accommodate both kinds of content.

The stack itself is a vector of stack nodes

```
vector<stackNode> stack;
```

or, even better, you can use a proper stack data structure from the C++ library.

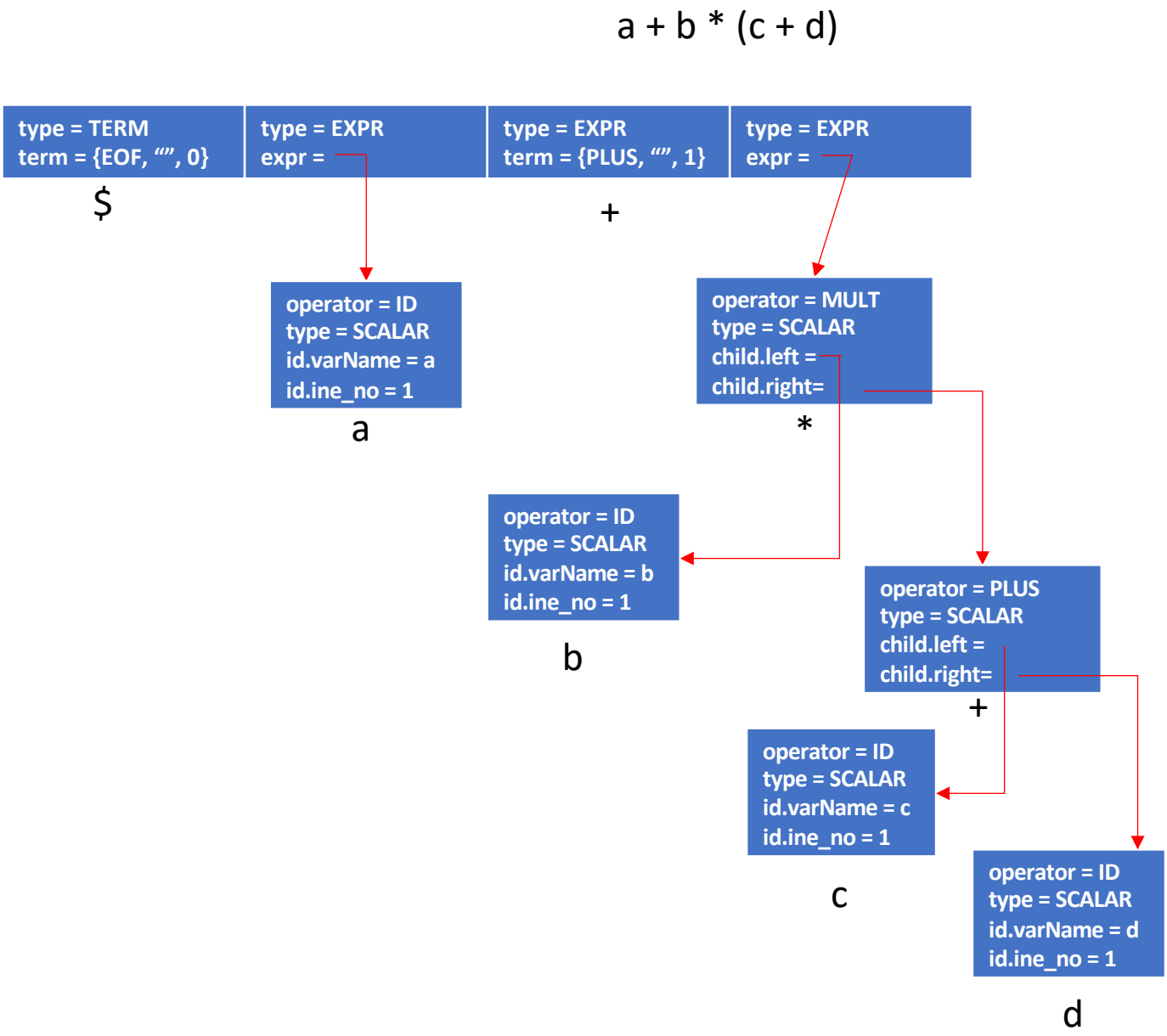
The treeNode struct can look something like this

```
Struct exprNode {
   int operator; // enum type: ID_OPER, PLUS_OPER, MINUS__OPER, DIV_OPER
                          //             ARRAY_ELEM_OPER, WHOLE_ARRAY_OPER
   int type;              // type of expression SCALAR, ARRAY, ARRAYDDECL, or ERROR
                          // these types are discussed later under type checking
   union {
      struct {                          // operator = ID
          String varName;
          int line_no;
      } id;

      struct {                          // operator = PLUS_OPER, MINUS_OPER
                                        //            MULT_OPET or ARRAY_ELEM_OPER
          struct treeNode *left;
          struct treeNode *right;
      } child;

      struct {                          // operator = WHOLE_ARRAY_OPER
           String arrayName;
          int line_no;
      } array;
   }
}
```

Warning do NOT cut and paste from the document. It is not complete

Stack Content Example

Example If we consider the execution from Notes_4, we have the following stack content on page 62 (assuming all variables are declared as SCALAR variable).



Notice how the parentheses are not included in the tree because they are really not needed. The parentheses are in the input to tell us how to group `c+d`. Once we parsed it and we stored in the abstract syntax tree (AST) correctly grouped, there is not need for parentheses.

`parse_expr()`: stack operations

So far, we discussed the structures that will be needed to represent the stack contents which can accommodate both kinds of content (terminals and expressions).

Now, we look at the stack operations that we need.

1. **Peek at the terminal closest to the top** this is either the top of the stack or just below the top of the stack

2. **Shift a token on the stack**

3. **Reduce** This one is more involved. There are two parts to this:

    1. You should write a function that pops elements from the stack until the top of stack has a terminal that is $<\cdot$ the last popped terminal.
    2. Check if the popped elements match the RHS of one of the rules.
    3. Build the abstract syntax tree after the reduction

I discuss each of these points on the next page

`parse_expr()`: Peek at the terminal closest to the top

The implementation of this is straightforward.

If the top of the stack is terminal, that is what the function returns

Otherwise, if the top of the stack is not a terminal, you can look at the entry just below the top of stack

Otherwise 

`parse_expr()`: Shift a token on the stack

This is also straightforward. When you determine that the token needs to be shifted, you need to consume the token from the input using `get_symbol()` and create a `stackNode` that contains the token information. The type of the node should be `TERM`. After you create the `stackNode`, you should push it on the stack.

`parse_expr()`: **REDUCE:** popping elements and matching righthand side of rules

Popping the elements

For this functionality, you need to implement a loop that pops the elements of the stack until the top of the stack is a terminal `top` such that table[`top`][`last popped terminal`] = <.

The popped elements are supposed to represent the righthand side of one of the rules for *expr*. So, we need

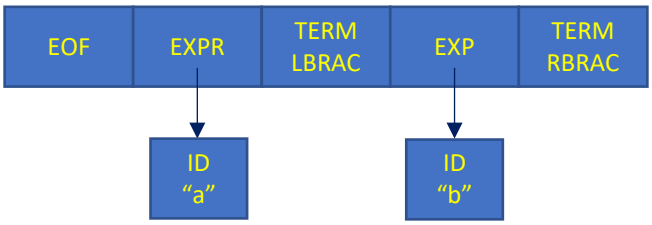
1. a way to represent the righthand sides of rules and
2. a way to compare the sequence of popped elements with the righthand sides and determine if the sequence of popped elements is a valid righthand side.

For each rule, you can store the righthand side as a vector of strings or the whole righthand side as a string (and maybe store it in a map). Here are some example representations

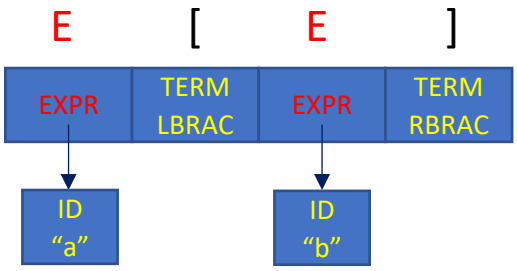
- *E* → *E*+*E* // "E+E"
- *E* → *E*-*E* // "E-E"
- *E* → *E*[*E*] // "*E*[E]"

As you pop the symbols, you can concatenate them together to obtain a string to be matched against the stored strings.

For example, if the stack state is the following



and the next token is PLUS, then you need to pop all the elements between <· and ·>. After the stack is popped, we end up with the following (actually depending on how you are storing what is being popped, you might end up with them in reverse order, but that should have no bearing on the solution as long as you are consistent with your order):

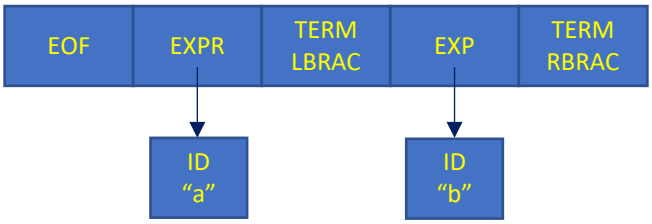


and the corresponding string is "*E*[E]". By comparing this string with the strings stored for each of the rules, you find that the popped match a righthand side of a rule, so we need to reduce it to *E*. This is where it gets a little interesting! You need a way to construct the abstract syntax tree of the reduced expression. Before doing that, we show on the next page the step in popping the stack elements.

`parse_expr()`: **REDUCE:** popping elements and matching righthand side of rules

Popping the elements

For example, if the stack state is the following



and the next token is PLUS, then you need to pop all the elements between $\lessdot$ and $\gtrdot$

The steps in popping the elements are as follows

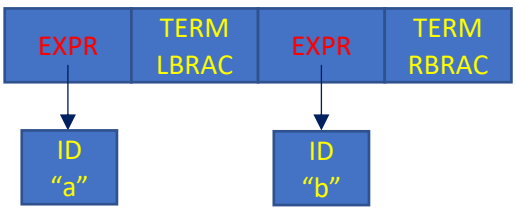
1. pop top of stack which is `RBRAC`
   since `RBRAC` is a `TERM`, we set `last_popped_term = RBRAC`
   since top of stack after popping is not a `TERM`, we continue popping

2. pop top of stack, which is `EXPR`
   since `EXPR` is a not a `TERM`, we don't change `last_popped_term`
   top of stack after popping `EXPR` is a `TERM` which is `LBRAC`
   since `LBRAC` $\doteq$ `RBRAC`, we continue popping

3. pop top of stack which is `LBRAC`
   since `LBRAC` is a `TERM`, we set `last_popped_term = LBRAC`
   since top of stack after popping is not a `TERM`, we continue popping

4. pop top of stack, which is `EXPR`
   since `EXPR` is a not a `TERM`, we don't change `last_popped_term`
   top of stack after popping `EXPR` is a `TERM` which is `EOF`
   since top of stack is a `TERM`, which is `EOF` and `EOF` $\lessdot$ `RBRAC`, we stop poping

After the stack is popped, we end up with the following (actually depending on how you are storing what is being popped, you might end up with them in reverse order, but that should have no bearing on the solution as long as you are consistent with your order):



and the corresponding string is "*E*[E]". This matches a righthand side of a rule, so we need to reduce it to *E*. This is where it gets a little interesting! You need a way to construct the abstract syntax tree of the reduced expression.

`parse_expr()`: Constructing the tree

For each reduction, you need to take the tree(s) of the popped elements (the righthand side of a rule) to make a tree of the righthand side of a rule. Here, we should note that the tree that represents the righthand side of a rule depends only on (1) the trees of the expressions on the righthand side, and (2) the rule being reduced.

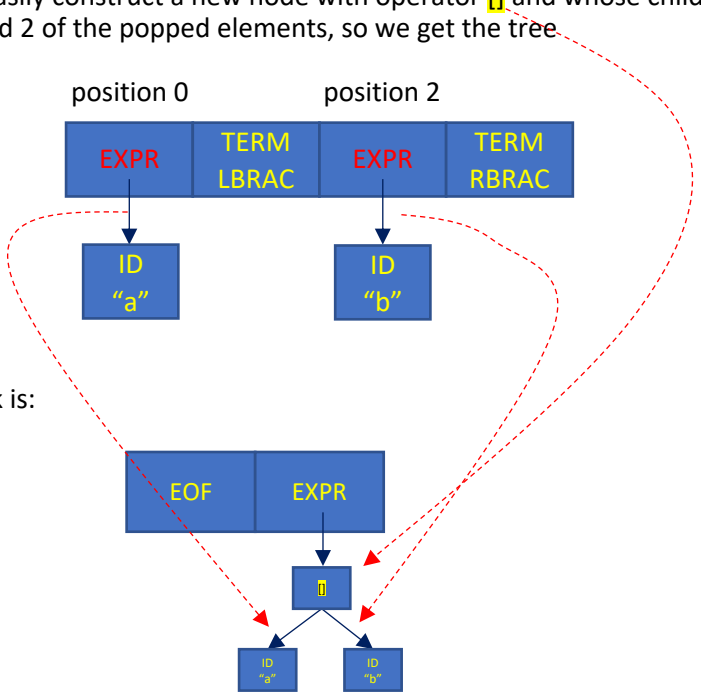
This require us to be able to determine in the vector of popped elements the positions that correspond to EXPR. While this can be checked with a loop, a loop is not needed. Since we already determined the the popped elements are a valid righthand side, the positions are uniquely determined by the righthand side that matched the popped elements. For that, you can store (hardcode), for each rule,

> the positions of the *expr* in the rule
> the operator of the rule (like `PLUS_OPER`, `MINUS_OPER`, …)
> whether the operator is binary or unary

For example, for the rule $E \rightarrow E\ [\ E\ ]$, the string representation of the RHS is "E[E]" and the positions that contain *E* are are 0 and 2. This says that in the vector of popped elements positions 0 and 2 contain expression nodes (with pointer to trees).

In addition to determining the positions, you need the determine the operator of the abstract syntax tree corresponding to the reduction. In the example above, the operator would be [] (appropriately represented in your data structures).

So, now we can easily construct a new node with operator [] and whose children are the trees from the positions 0 and 2 of the popped elements, so we get the tree



and the new stack is:

You can have a function that is provided with a vector of popped elements and positions and an operator and returns a new tree. This function will be able to handle all binary operators together (but type checking would need different handling for each operator)

`parse_expr()`: Type Checking

Type checking is relatively easy (compared to constructing the tree).

The idea is to include in each node of the tree a type which is one of the **array**, **scalar**, **error** or **arraydecl**. To understand why **arraydecl** is needed, we need to first understand what the `type` field is.

The `type` field in a node is used to store the type of the whole subtree whose root is the given node. This requires special handling for arrays.

Consider the example `a[b]` that we have seen on the previous page. The `left child` is an expression that is a primary that is an `ID`. If `a` is declared as an array, the expression consisting of `a` by itself has no type on its own, according to the typing rules. Still, to enable all the type checking for a tree node to be done simply by looking at the operator and the types of the children, we need to assign a type for the case in which `a` is simply a declared array.

We need to use **arraydecl** is for IDs that are declared as array, which do no have a type according to the code specification, when they appear in an expression. Still, that would be useful for type checking. For example, if we have an ID of type **arraydecl** and we apply the operator [.] to it, we get an expression of type **array**.

Once all is in place, you can do type checking recursively on the tree itself. The base case if for nodes that contain IDs or NUM in which case:

1. The type is arraydecl if the ID is declared as array
2. The type is scalar if the ID is declared as scalar
3. The type is scalar in the case of NUM

For the induction, you have node with an operator and one or two children nodes each have a type that is already calculated (recursively), so the type checking is now a simple case analysis based on the operator and the types of the children.

Type checking for variable-access is similar. It depends on the ID in case the variable access is simply an ID or ID[.] and on the ID and the *expr* in the case of a variable access ID[*expr*].

`parse_expr()`: Type Checking

Type checking is relatively easy (compared to constructing the tree).

The idea is to include in each node of the tree a type which is one of the **array**, **scalar**, **error** or **arraydecl**.

arraydecl is for IDs that are declared as array, which do no have a type according to the code specification, when they appear in an expression. Still, that would be useful for type checking. For example, if we have an ID of type **arraydecl** and we apply the operator [.] to it, we get an expression of type **array**.

Once all is in place, you can do type checking recursively on the tree itself. The base case if for nodes that contain IDs or NUM in which case:

1. The type is arraydecl if the ID is declared as array
2. The type is scalar if the ID is declared as scalar
3. The type is scalar in the case of NUM

For the induction, you have node with an operator and one or two children nodes each have a type that is already calculated (recursively), so the type checking is now a simple case analysis based on the operator and the types of the children.

Type checking for variable-access is similar. It depends on the ID in case the variable access is simply an ID or ID[.] and on the ID and the *expr* in the case of a variable access ID[*expr*].

Line Numbers

Remember that line numbers are available to you in the Token struct. You can take that information and include it in the abstract syntax tree. This way, if the top of the tree has type error (for an expression tree) or type mismatch (for an assignment), you will have the line number handy.

Printing the abstract syntax tree

The abstract syntax tree should be printed in breadth first manner. This can be easily done recursively. Jacob plan on reviewing trees and breadth first traversal in the C++ session on Wednesday.

Code generation

No implementation suggestions are provided.