Assignment 2 - Parsing C Expressions

We need to get started somewhere parsing the C language. We can think of the grammar in 5 broad categories:

1) declarations

2) function definitions

3) statements (including blocks)

4) expressions

5) miscellaneous constructs such as abstract types

A good place to start is expressions. In this assignment, you'll construct the .y file to parse C language expressions, and transform them into Abstract Syntax Tree notation. You'll also write a decoder for your AST data structures which allows them to be visualized, e.g. by printing them out in plain text.

The Expression Grammar

The full grammar for C expressions can be found in H&S or the ISO C standard. Note that both sources use BNF notation and break the expressions down into individual operator precedence levels. You certainly can use this form, or you can make the grammar more compact by using Bison's operator precedence features. E.g. you could reduce this to primary expressions, postfix expressions, unary expressions, binary expressions, assignment expressions and ternary expressions. Any approach you use is fine, as long as it correctly parses the language!

To simplify your .y file at this early stage, we can make the following assumptions:

1) The top-level of the grammar is a list of expression statements. There are no other kinds of statements, nor are there function definitions or declarations. So your input looks like:

```
a=b+c/3;
z?(b/=3):c++;
etc;  /* This is a valid expression statement! */
```

2) We don't have abstract type names yet so you can't implement the casting operator or the sizeof operator taking a type name (but you can implement sizeof taking an ordinary expression).

3) We don't have declarations, scopes or symbol tables, so it is OK to represent identifiers as just their names for now.

4) Forget about "generic selection" expressions.

5) Don't worry about compound literals.

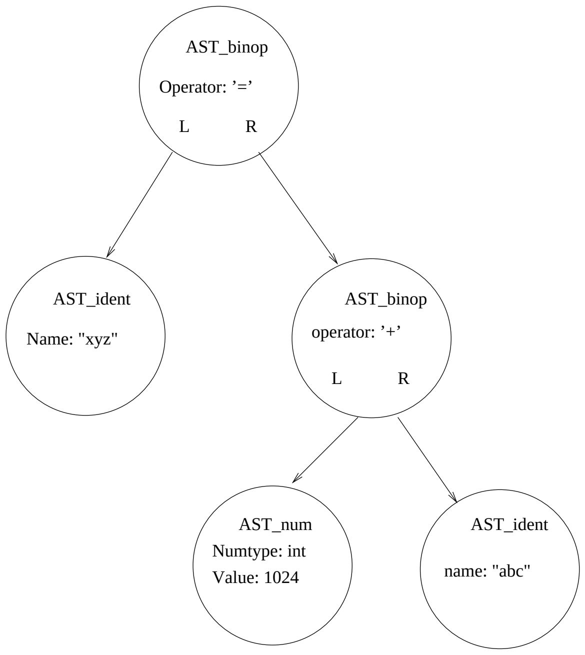
6) The `alignof` keyword/operator has undergone many permutations since it was introduced as an optional feature in C11. It can be _alignof, _Alignof, __alignof, or just plain alignof. All of the official ISO references say this operator can only take a type_name, but it is more logical that it should work the same as sizeof, which can

take a type_name or an expression. In fact, GCC allows alignof(expression). Since we do not have type_name in our grammar yet, you can only implement alignof(expression). But don't over-exert yourself for this minor feature.

Abstract Synax Trees

As we get further into the class, we'll go into more detail about a type of intermediate representation known as Abstract Syntax Trees. Simply put, this data structure graphs the "abstract" relationships between the elements of the language as they were parsed. For example:

```
xyz=(1024+abc);
```



We see that this expression can be represented as a graph where each node has a specific type and contains fields that pertain to its purpose. The AST_binop node contains left and right pointers to its two operands, plus a field to contain the operator. In this example, I envision using the token code to represent the operator but the actual design and implementation is quite up to you.

Notice that the parenthesized expression does not appear in the AST. That's what makes it an "abstract" syntax tree, as opposed to the literal parse tree. Parentheses serve no semantic purpose. They are in the language for syntactic reasons only, to override the default order of operations. Once we parse the expression, there is no reason to complicate the AST with the memory of the parentheses!

You'll also notice that for identifiers and numbers, the fields contained in those nodes are basically the same as what you did in assignment #1 (lexer). As we get deeper into the project, this might require more refinement.

Printing out AST

We need a way to print out the AST to make sure it is actually constructed correctly! Here is a simple, text-based way:

```
ASSIGNMENT
  IDENT xyz
  BINARY OP +
    NUM (numtype=int) 1024
    IDENT abc
```

Notice that indentation is used to represent the depth levels of the tree. Matching it up to the graphical example above it should be readily apparent how to recursively descend an AST and keep track of the indentation levels.

There is no "right format" and students are free to introduce braces or brackets, or use a totally different format. The only requirement is that your printout presents all of the information in the AST including the relationships between elements.

The ability to print out ASTs is not something you will throw away after assignment 2. It will be a useful debugging tool as we get deeper into the project. So, don't write throwaway code!

Some students in the past have spent a lot of time producing a tool to create a graphical representation of the AST. While this can be very pleasing, it can also be a sinkhole of time. Don't get stuck in it!

Shortcuts/Equivalencies

There are some C language constructs which are defined to be exactly equivalent to other constructs. It will be helpful to apply those equivalencies during the parsing phase to minimize the amount of redundant (as in, literally, exactly the same) code both now during AST construction, and later during type analysis and IR generation from the AST.

The first is array indexing: a[b] is exactly the same as *(a+b) and while it may seem that the latter form is more complicated, as it turns out we'll need to do "pointer arithmetic" for arrays anyway, so I suggest turning array indexing into the equivalent pointer operation during parsing.

Another example is the -> indirect structure member operator. `a->m` is exactly the same as `(*a).m`

Compound assignment operators, e.g. a+=b, may appear to be equivalent to a=a+b. However, there is a subtle trap: if the subexpression a includes a side effect or function call. E.g. a[f(x)]+=3; Clearly the function f must only be called once when evaluating this expression.

Pre-increment/decrement operators can be replaced: `--a` is exactly the same as `(a -= 1)` and likewise for `++a`. This is because any assignment operator is also an rvalue. But we can't do the same for the post operators: `a++` is certainly not `(a += 1 )` when you consider the impact to the expression in which it is placed.

Implementations of AST

There are so many ways to code ASTs that it would be pointless to try to list them here. Because the AST node is variadic, you need a "tag" or "discriminator" to tell you what is inside. In traditional C, this could be implemented like this:

```
struct astnode {
          int nodetype;
          union astnodes {
                    struct astnode_binop binop;
                    struct astnode_num  num;
                    struct astnode_ident ident;
                    /* etc. */
          } u;
};
```

This has the slight ugliness of having to involve the extraneous reference to element `u` whenever you need to access an AST node. GCC has always had an extension called "anonymous or unnamed struct/union members" which was ultimately made part of C-11:

```
struct astnode {
          int nodetype;
          union {
                    struct astnode_binop binop;
                    struct astnode_num num;
                    struct astnode_ident ident;
          };
}
```

The union members such as `binop` can be accessed directly, e.g. `p->binop.left` where p is a `struct astnode *`. Note that section 6.7.2.1.13 of C11 (6.7.3.2.15 in C23) describes this in a very confusing way. The members binop, num, etc. are not each substructs of struct astnode. Rather, this is just a syntax shortcut to avoid giving the union a name. Note that the anonymous union must be tagless and that the member names of the anonymous union can not conflict with the member names of the enclosing struct astnode. We'll have more to say about struct/union definitions in Unit 4.

I've also seen students approach AST nodes like this:

```
struct astnode {
          int nodetype;
          struct astnode *pointers[SOME_REASONABLE_NUMBER];
          int nchild;
          union astnode_values {
                    struct astnode_binop binop;
                    /* etc. */
          } val;
};
```

Here the tree structure of the AST is exposed at the top level rather than being inside each individual ast node type.

Here is yet another way using an old dirty trick called "overlapping struct definitions":

```
/* Remember, these are just syntax examples, your actual AST nodes may differ */
struct astnode_binop {
          int nodetype;
          int operator;
          union astnode *left,*right;
};

struct astnode_num {
          int nodetype;
          int number;
};


union astnode {
          struct astnode_generic {int nodetype;} generic;
          struct astnode_binop binop;
          struct astnode_num num;
          /* etc.*/
}
```

Here we rely on the fact that structs will always be laid out consistently, and define a series of structs that share the first element (the node type tag). The C11 standard (6.5.2.3.6, also 6.5.3.4.6 in C23) now explicitly recognizes that this "old trick" is guaranteed to work.

Some Bison Tricks

You are definitely going to want to make your Bison grammar fully typed and use the `%union` and `%type` directives. To create a nested AST, it will be very helpful if you make the type of the grammar symbols such as `expr` be a pointer to an AST node. Then you can create additional AST nodes as needed and chain together pointers. E.g.

```
%union {
          struct astnode *astnode_p;
}

%type <astnode_p> binary_expr
%%
binary_expr:
          binary_expr '+' binary_expr {
                              $$=astnode_alloc(AST_binop);
                              struct astnode_binop *n=(void *)$$;
                              n->operator='+';
                              n->left=$1;
                              n->right=$3;
                              }
```

The above is just a representative snippet and your code might be significantly different, depending on how you implement ASTs and how you structure your grammar.

Testing / Submission

Your program should accept a list of expression statements. After it sees each one, it should print out the AST representation of that statement. Alternatively, you could have an AST construction for a "LIST" and build the list of expression ASTs for the entire input, and output everything upon reaching EOF.

I have included some test cases but they are by no means exhaustive.

When you submit your work, include the test cases that you ran and the output they produced. If there are things in the expression grammar that aren't working at the time of submission, note that.