Assignment 2 - Parsing C Expressions

We need to get started somewhere parsing the C language. We can think of the grammar as broken into 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 function definitions, declarations, or other kinds of statements, for now. So your input looks like:

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

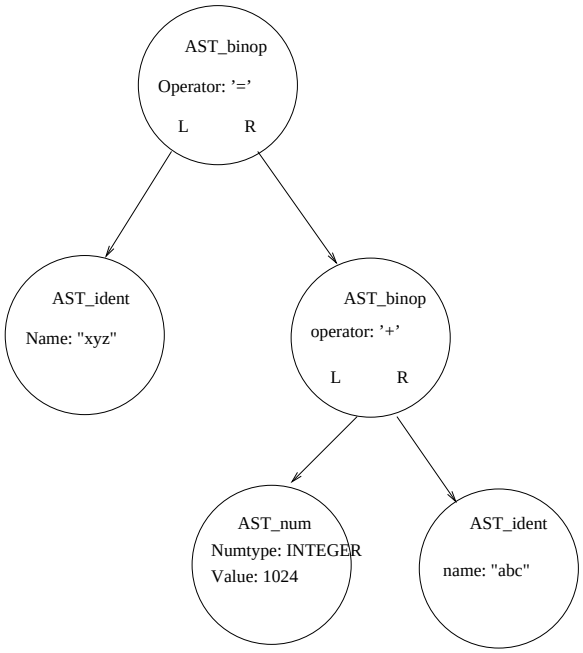
2) We don't have abstract type names yet so you can't implement the casting operator or the sizeof operator taking a type name (but you can implement sizeof taking an ordinary expression).

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

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, they are a data structure which 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 langauge 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.

Shortcuts/Equivalencies

Certain language constructs are semantically equivalent to certain other constructs, which can reduce the amount of code and complexity needed in your AST. For example, the expression `a+=b` is exactly equivalent to `a=a+b`. It is advisable to eschew

having a += AST node because that will just be extra code that you need to carry around for the rest of your compiler, including code generation! Instead, apply the transformation and construct the AST as if it had been parsed the second way.

There will be other similar opportunities such as:

```
a[b]   ===>  *(a+b)
a->b   ===>  (*a).b
--a    ===>  (a=a-1)          /* But not a++ nor a-- !!! */
++a    ===>  (a=a+1)
```

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 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. But if you use the GCC extension known as "blind /anonymous unions" then you can avoid that. This extension has now made it into the C-11 standard so any modern compiler would support it. I've also seen students approach it 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. There are many, many different ways to do this. Here is a slightly dirty way:

```
struct astnode_binop {
          int nodetype;
          int operator;
          union astnode *left,*right;
};

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).

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 {
                              $$=astnode_alloc(AST_binop);
                              struct astnode_binop *n=$$;
                              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.

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.