Introduction to Compilers and Language Design

You can find the latest version of the PDF edition, and purchase inexpensive hardcover copies at this website:
`http://compilerbook.org`

Draft version: February 2, 2017

Chapter 6 – The Abstract Syntax Tree

6.1 Overview

The Abstract Syntax Tree (AST) is an important internal data structure that represents the primary structure of a program. The AST is the starting point for semantic analysis of a program. It is "abstract" in the sense that the structure leaves out the particular details of parsing: the AST does not care whether a language has prefix, postfix, or infix expressions. (In fact, the AST we describe here can be used to represent most procedural languages.)

For our project compiler, we will define an AST in terms of five C structures representing declarations, statements, expressions, types, and parameters. While you have certainly encountered each of these terms while learning programming, they are not always used precisely in practice. This chapter will help you to sort those items out very clearly.

For each kind of element in the AST, we will give an example of the code and how it is constructed. Because each of these structures potentially has pointers to each of the other types, it is necessary to preview all of them before seeing how they work together.

Once you understand all of the elements of the AST, we finish the chapter by demonstrating how the entire structure can be created automatically through the use of the Bison parser generator.

6.2 Declarations

A complete C-Minor program is a sequence of declarations. Each declaration states the existence of a variable or a function. A variable declaration may optionally give an initializing value if none is given, it is given a default value of zero. A function declaration may optionally give the body of the function in code; if no body is given, then the declaration serves as a prototype for a function declared elsewhere.

For example, the following are all valid declarations:

```
b: boolean;
s: string = "hello";
f: function integer ( x: integer ) = { x * x };
```

A declaration is represented by a `decl` structure that gives the name, type, value (if an expression), code (if a function), and a pointer to the next declaration in the program:

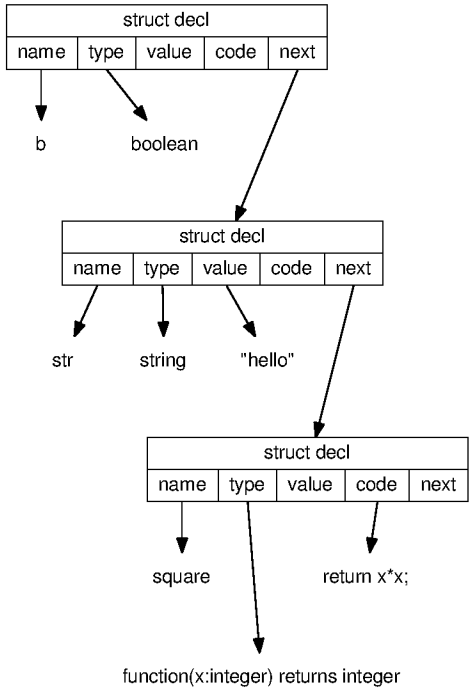
```
struct decl {
    char *name;
    struct type *type;
    struct expr *value;
    struct stmt *code;
    struct decl *next;
};
```

Because we will be creating a lot of these structures, you will need a factory function that allocates a structure and initializes its fields, like this:

```
struct decl * decl_create( char *name,
                           struct *type,
                           struct expr *value,
                           struct stmt *code )
{
    struct decl *d = malloc(sizeof(*d));
    d->name = name;
    d->type = type;
    d->value = value;
    d->code = code;
    d->next = 0;
    return d;
}
```

(You will need to write similar code for statements, expressions, etc, but we won't keep repeating it here.)

The three declarations on the preceding page can be represented graphically as a linked list, like this:



Note that some of the fields point to nothing: these would be represented by a null pointer, which we omit for clarity. Also, our picture is incomplete and must be expanded: the items representing types, expressions, and statements are all complex structures themselves that we must describe.

6.3 Statements

The body of a function consists of a sequence of statements. A statement indicates that the program is to take a particular action, in the order specified, such as computing a value, performing a loop, or choosing A statement can also be a declaration of a local variable. Here is the `stmt` structure:

```
struct stmt {
    stmt_t kind;
    struct decl *decl;
    struct expr *init_expr;
    struct expr *expr;
    struct expr *next_expr;
    struct stmt *body;
    struct stmt *else_body;
    struct stmt *next;
};
```

```
typedef enum {
    STMT_DECL,
    STMT_EXPR,
    STMT_IF_ELSE,
    STMT_FOR,
    STMT_PRINT,
    STMT_RETURN,
    STMT_BLOCK
} stmt_t;
```

The `kind` field indicates what kind of statement it is:

- `STMT_DECL` indicates a (local) declaration, and the `decl` field will point to it.
- `STMT_EXPR` indicates an expression statement and the `expr` field will point to it.
- `STMT_IF_ELSE` indicates an if-else expression such that the `expr` field will point to the control expression, the `body` field to the statements executed if it is true, and the `else_body` field to the statements executed if it is false.
- `STMT_FOR` indicates a for-loop, such that `init_expr`, `expr`, and `next_expr` are the three expressions in the loop header, and `body` points to the statements in the loop.
- `STMT_PRINT` indicates a `print` statement, and `expr` points to the expressions to print.
- `STMT_RETURN` indicates a `return` statement, and `expr` points to the expression to return.
- `STMT_BLOCK` indicates a block of statements inside curly braces, and `body` points to the contained statements.

6.4 Expressions

Expressions are implemented much like the simple expression AST shown in Chapter 5. The difference is that we need many more binary types: one for every operator in the language, including arithmetic, logical, comparison, assignment, and so forth. We also need one for every type of leaf value, including variable names, constant values, and so forth. The `name` field will be set for `EXPR_NAME`, the `integer_value` field for `EXPR_INTEGER_LITERAL`, and so on. You may need to add values and types to this structure as you expand your compiler.

```
struct expr {
    expr_t kind;
    struct expr *left;
    struct expr *right;

    const char *name;
    int integer_value;
    const char * string_literal;
};
```

```
typedef enum {
    EXPR_ADD,
    EXPR_SUB,
    EXPR_MUL,
    EXPR_DIV,
    ...
    EXPR_NAME
    EXPR_INTEGER_LITERAL,
    EXPR_STRING_LITERAL,
} expr_t;
```

As before, you should create a factory for a binary operator:

```
struct expr * expr_create( expr_t kind,
                           struct expr *L, struct expr *R );
```

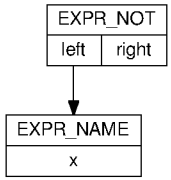
And then a factory for each of the leaf types:

```
struct expr * expr_create_name( const char *name );
struct expr * expr_create_integer_literal( int i );
struct expr * expr_create_boolean_literal( char c );
struct expr * expr_create_char_literal( int b );
struct expr * expr_create_string_literal( const char *str );
```

Note that you can store the integer, boolean, and character literal values all in the `integer_value` field.

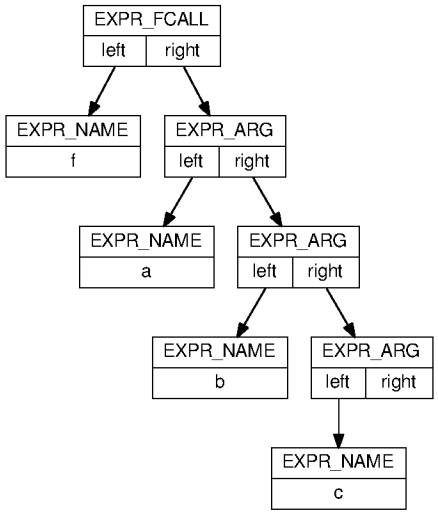
A few cases deserve special mention. Unary operators like logical-not typically have their sole argument in the `left` pointer:

```
!x
```



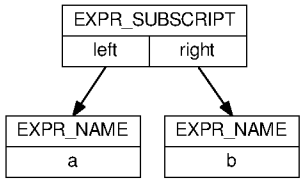
A function call is constructed by creating an `EXPR_CALL` node, such that the left-hand side is the function name, and the right hand side is an unbalanced tree of `EXPR_ARG` nodes. While this looks a bit awkward, it allows us to express a linked list using a tree, and will simplify the handling of function call arguments on the stack during code generation.

```
f(a,b,c)
```



Array subscripting is treated like a binary operator, such that the name of the array is on the left side of the `EXPR_SUBSCRIPT` operator, and an integer expression on the right:

```
a[b]
```



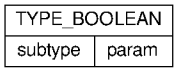
6.5 Types

A `type` structure encodes the type of every variable and function mentioned in a declaration. Primitive types like `integer` and `boolean` are expressed by simply setting the `kind` field appropriately, and leaving the other fields null. Compound types like `array` and `function` are built by connecting multiple `type` structures together.

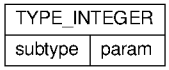
```
typedef enum {
    TYPE_VOID,
    TYPE_BOOLEAN,
    TYPE_CHARACTER,
    TYPE_INTEGER,
    TYPE_STRING,
    TYPE_ARRAY,
    TYPE_FUNCTION
} type_t;

struct type {
    type_t kind;
    struct type *subtype;
    struct param_list *params;
};

struct param_list {
    char *name;
    struct type *type;
    struct param_list *next;
};
```

For example, to express a basic type like a boolean or an integer, we simply create a standalone `type` structure, with `kind` set appropriately, and the other fields null:

`boolean`

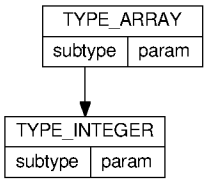


`integer`



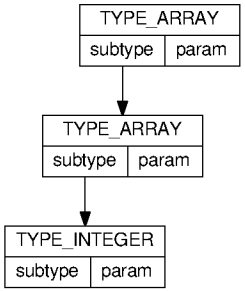
To express a compound type like an array of integers, we set `kind` to `TYPE_ARRAY` and set `subtype` to point to a `TYPE_INTEGER`:

`array [] integer`



These can be linked to arbitrary depth, so to express an array of array of integers:

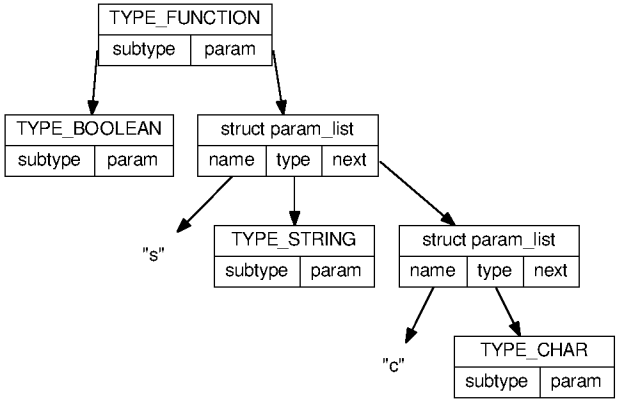
`array [] integer`



To express the type of a function, we use `subtype` to express the return type of the function, and then connect a linked list of `param_list` nodes to describe the name and type of each parameter to the function.

For example, here is the type of a function which takes two arguments and returns an integer:

```
function integer (s:string, c:char)
```



6.6 Putting it All Together

Now that you have seen each individual component, let's see how a complete C-Minor function would be expressed as an AST:

```
compute: function integer ( x:integer ) = {
        i: integer;
        total: integer = 0;
        for(i=0;i<10;i++) {
                total = total + i;
        }
        return total;
}
```