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: December 12, 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:

- A **declaration** states the name, type, and value of a symbol so that it can be used in the program. Symbols included items such as constants, variables, and functions.
- A **statement** indicates an action to be carried out that changes the state of the program. Examples include loops, conditionals, and function returns.
- An **expression** is a combination of values and operations that is **evaluated** according to specific rules and yields a **value** such as an integer, floating point, or string. In some programming languages, an expression may also have a **side effect** that changes the state of the program.

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 ) = { return 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:

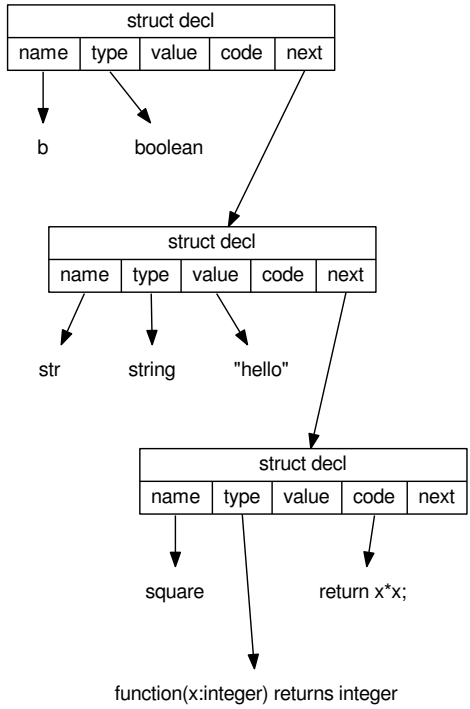
```
struct decl {
    char *name;
    struct type *type;
    struct expr *value;
    struct stmt *code;
    struct decl *next;
};
```

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 *next )
{
    struct decl *d = malloc(sizeof(*d));
    d->name = name;
    d->type = type;
    d->value = value;
    d->code = code;
    d->next = next;
    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 between branches of an alternative. 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:

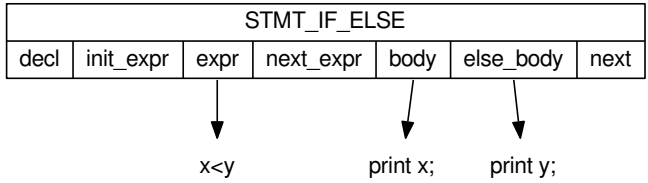
- `STMT_DECL` indicates a (local) declaration, and the `decl` field will point to it.
- `STMT_EXPR` indicates an expression statement and the `expr` field will point to it.
- `STMT_IF_ELSE` indicates an if-else expression such that the `expr` field will point to the control expression, the `body` field 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.

And, as we did with declarations, we require a function `stmt_create` to create and return a statement structure:

```
struct stmt * stmt_create( stmt_kind_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 );
```

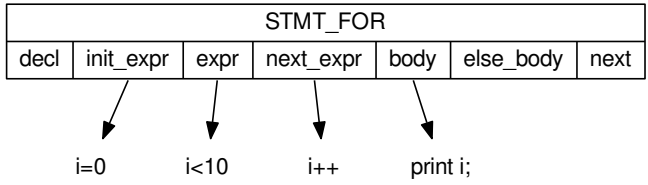
This structure has a lot of fields, but each one serves a purpose and is used when necessary for a particular kind of statement. For example, an if-else statement only uses the `expr`, `body`, and `else_body` fields, leaving the rest null:

```
if( x<y ) print x; else print y;
```



A for-loop uses the three `expr` fields to represent the thre parts of the loop control, and the `body` field to represent the code being executed:

```
for(i=0;i<10;i++) print i;
```



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 );
```

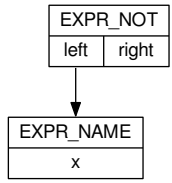
And then a factory for each of the leaf types:

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

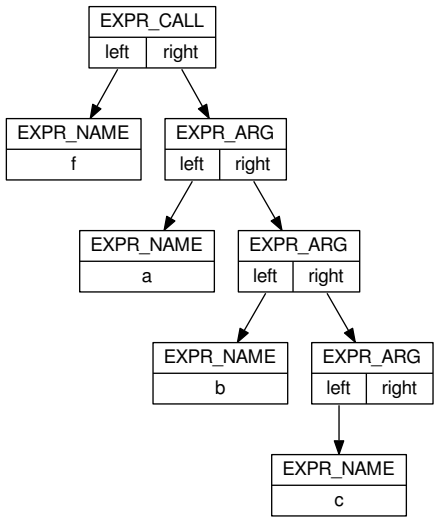
A few cases deserve special mention. Unary operators like logical-not typically have their sole argument in the `left` pointer:

```
!x
```



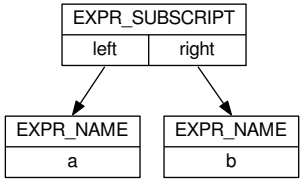
A function call is constructed by creating an `EXPR_CALL` node, such that the left-hand side is the function name, and the right hand side is an unbalanced tree of `EXPR_ARG` nodes. While this looks a bit awkward, it allows us to express a linked list using a tree, and will simplify the handling of function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



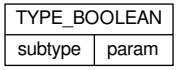
6.5 Types

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.

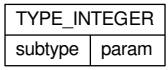
```
typedef enum {
    TYPE_VOID,
    TYPE_BOOLEAN,
    TYPE_CHARACTER,
    TYPE_INTEGER,
    TYPE_STRING,
    TYPE_ARRAY,
    TYPE_FUNCTION
} type_t;

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`

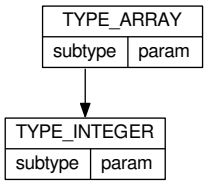


`integer`



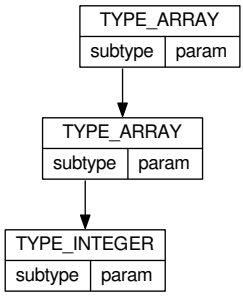
To express a compound type like an array of integers, we set `kind` to `TYPE_ARRAY` and set `subtype` to point to a `TYPE_INTEGER`:

`array [] integer`



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

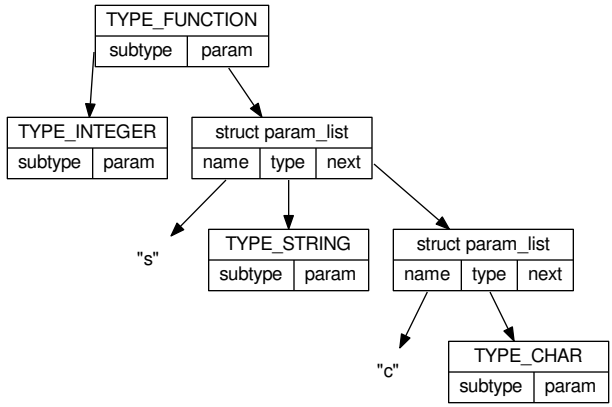
`array [] array [] integer`



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)
```



Note that the type structures here let us express some complicated and powerful higher order concepts of programming. By simply swapping in complex types, you can describe an array of ten functions, each returning an integer:

```
a: array [10] function integer ( x: integer );
```

Or how about a function that returns a function?

```
f: function function integer (x:integer) (y:integer);
```

Or even a function that returns an array of functions!

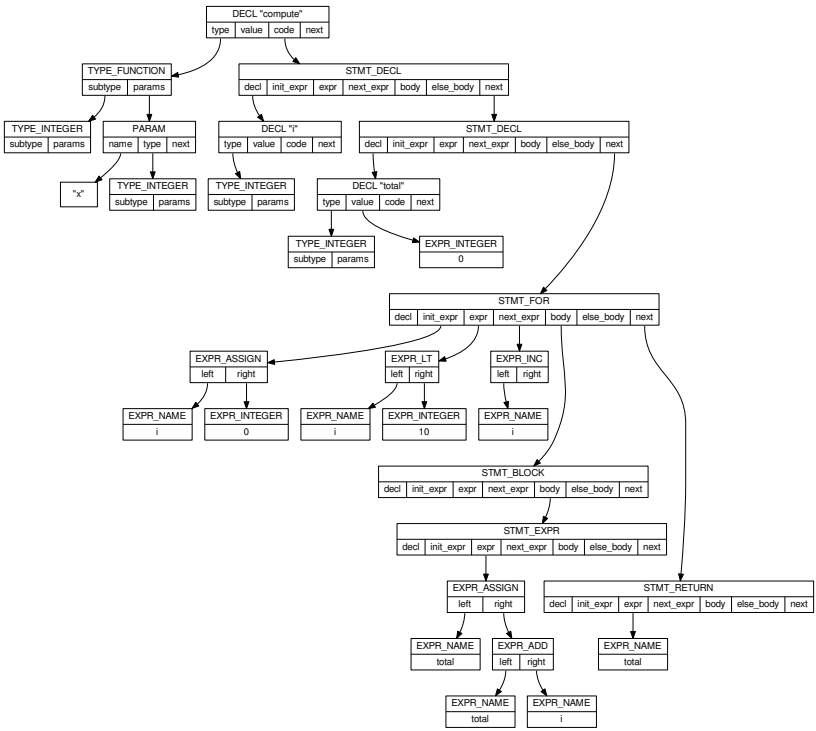
```
g: function array [10]
       function integer (x:integer) (y:integer);
```

While the C-Minor type system is capable of *expressing* these ideas, these combinations will be rejected later in typechecking, because they require a more dynamic implementation than we are prepared to create. If you find these ideas interesting, then you should read up on functional languages such as Scheme and Haskell.

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;
}
```



6.7 Building the AST

With the functions created so far in this chapter, we could, in principle, construct the AST manually in a sort of nested style,. For example, the following code represents a function called `square` which accepts an integer `x` as a parameter, and returns the value `x*x`:

```
d = decl_create(
    "square"
    type_create(TYPE_FUNCTION,
        type_create(TYPE_INTEGER,0,0),
        param_list_create(
            "x",
            type_create(TYPE_INTEGER,0,0))),
    0,
    stmt_create(STMT_RETURN,0,0,
        expr_create(EXPR_MUL,
            expr_create_name("x"),
            expr_create_name("x")),
        0,0,0,0)
);
```

Obviously, this is no way to write code! Instead, we want our parser to invoke the various creation functions whenever they are reduced, and then hook them up into a complete tree. Using an LR parser generator like Bison, this process is straightforward. (Here I will give you the idea of how to proceed, but you will need to figure out many of the details in order to complete the parser.)

At the top level, a C-Minor program is a sequence of declarations:

```
program : decl_list
          { parser_result = $1; }
        ;
```

Then, we write rules for each of the various kinds of declarations in a C-Minor program:

```
decl : name TOKEN_COLON type TOKEN_SEMI
       { $$ = decl_create($1,$3,0,0,0); }
     | name TOKEN_COLON type TOKEN_ASSIGN expr TOKEN_SEMI
       { $$ = decl_create($1,$3,$5,0,0); }
     | /* and more cases here */
       . . .
     ;
```

Since each `decl` structure is created separately, we must connect them together in a linked list formed by a `decl_list`. This is most easily done

by making the rule right-recursive, so that `decl` on the left represents one declaration, and `decl_list` on the right represents the remainder of the linked list. The end of the list is a null value provided when `decl_list` produces ϵ.

```
decl_list : decl decl_list
            { $$ = $1; $1->next = $2; }
          | /* epsilon */
            { $$ = 0; }
          ;
```

For each kind of statement, we create a `stmt` structure that pulls out the necessary elements from the grammar.

```
stmt : TOKEN_IF TOKEN_LPAREN expr TOKEN_RPAREN stmt
       { $$ = stmt_create(STMT_IF_ELSE,0,0,$3,0,$5,$0,0); }
     | TOKEN_LBRACE stmt_list TOKEN_RBRACE
       { $$ = stmt_create(STMT_BLOCK,0,0,0,0,$2,0,0); }
     | /* and more cases here */
       . . .
     ;
```

Proceed in this way down through each of the grammar elements of a C-Minor program: declarations, statements, expressions, types, parameters, until you reach the leaf elements of literal values and symbols, which are handled in the same way as in Chapter 5.

There is one last complication: What, exactly is the semantic type of the values returned as each rule is reduced? It isn't a single type, because each kind of rule returns a different data structure: a declaration rule returns a `struct decl *`, while an identifier rule returns a `char *`. To make this work, we inform Bison that the semantic value is the union of all of the types in the AST:

```
%union {
        struct decl *decl;
        struct stmt *stmt;
        . . .
        char *name;
};
```

And then indicate the specific subfield of the union used by each rule:

```
%type <decl> program decl_list decl . . .
%type <stmt> stmt_list stmt . . .
. . .
%type <name> name
```

6.8 Exercises

1. Write a complete LR grammar for C-Minor and test it using Bison. Your first attempt will certainly have many shift-reduce and reduce-reduce conflicts, so use your knowledge of grammars from Chapter 4 to rewrite the grammar and eliminate the conflicts.

2. Write the AST structures and generating functions as outlined in this chapter, and manually construct some simple ASTs using nested function calls as shown above.

3. Add new functions `decl_print()`, `stmt_print()`, etc. that print the AST back out so you can verify that the program was generated correctly. Make your output nicely formatted using indentation and consistent spacing, so that the code is easily readable.

4. Add the AST generator functions as action rules to your Bison grammar, so that you can parse complete programs, and print them back out again.

5. Add new functions `decl_translate()`, `stmt_translate()`, etc. that output the C-Minor AST in a different language of your own choosing, such as Python or Java or Rust.

6. Add new functions that emit the AST in a graphical form so you can "see" the structure of a program. One approach would be to use the Graphviz DOT format: let each declaration, statement, etc be a node in a graph, and then let each pointer between structures be an edge in the graph.