1 Introduction

In this lab assignment, we are going to implement the first phase of the compiler: *lexing* and *parsing*. The goal of this phase is to produce an *abstract syntax tree* (AST). An AST is a data structure that represents the program in a file in a simplified *tree form*. Comments, white space, parenthesization have been removed. The AST representation makes it much easier for the rest of the compiler to examine and interpret the program.

The front end of a compiler has three parts:

1. **Lexing** separates the text in an input file into *tokens* that represent things like numbers, identifiers, special characters, keywords, etc. It also removes comments and whitespace. We will implement the lexer using the `ocamllex` program.
2. **Parsing** produces the AST from the sequence of tokens returned by the lexer. The parser is specified as a *context-free grammar* that includes a *semantic action* for each production in the grammar. We will implement the parser using the `ocamlyacc` program.
3. **Semantic analysis** generates intermediate code from the AST, also performing type checking. This will be the topic of Lab 3.

2 Source language

First, we have to define the language we are compiling. We'll be implementing a subset of C called *functional C*. The complete syntax is described in K&R, but we won't be using the whole language. The main parts that are missing are the following. All of these are implementable, but we will omit them to keep the compiler simpler.

- no `switch` statements,
- no labels or `goto` statements,
- no declaration modifiers like `long`, `unsigned`, `static`, etc.,
- no `void` type,
- no explicit coercions, like `(int *) malloc(4)`,
- no functions with a variable number of arguments, like `printf`,
- no array or `struct` initializers,
- no `union` types,
- no `do` loops.

We also impost some additional semantic restrictions to make things easier.

- Structures must be defined before being used. That is, the following code is acceptable:

```
struct t {
   int i;
};

struct t x;
```

However, the following code would be rejected if `struct t` is not defined.

```
struct t x;
```

If you like, you may also choose to reject the following code.

```
struct t *x;
```

- We will treat `struct` definitions as `typedefs` (like in C++). The following code is acceptable.

```
struct IntList {
   int i;
   IntList *next;
};
```

- No `struct` definitions in parameter lists. The following code may be rejected:

```
int f(struct t { int i; } x)
{
   return x.i;
}
```

However, we will be adding two features to make the language (semi) *functional*. First, we will allow arbitrary nesting of function definitions. We may, or may not, allow inner functions to be passed as arguments and returned from functions. We'll see as the term goes along. Second, every expression will return a value (although the value may be useless). In particular, we will not differentiate between the `if` statement, and the *test*?*expr* : *expr* expressions. Statement blocks like $\{s_1; s_2; \ldots; s_n\}$ will evaluate the statements $s_1, \ldots, s_n$ in order and return the value of the last one. The empty $\{\}$ statement block will return 0.

We now define the language in more detail, based on K&R.

2.1 Lexical converntions

2.1.1 Comments

Comments begin with the characters `/*` and are terminated with `*/`. Any text is allowed in a comment, but comments can't be nested.

2.1.2 Identifiers

Identifiers (for variable, type, and function names) are a sequence of letters and digits, starting with a letter. The undescore _ counts as a letter.

2.1.3 Keywords

The following words are used as keywords.

break	char	else	float
for	if	int	return
sizeof	struct	typedef	while

2.1.4 Constants

- *character* constants have the form `'c'` where `c` is a single character, or an escaped character:

  - `\n` is a newline
  - `\t` is a horizontal tab char
  - `\'` is a single quote char
  - `\"` is a double quote char

  You may implement the other escape sequences if you like.

- *integers* have the following form:

  decimal `['1'-'9']['0'-'9']*`

  octal `'0' ['0'-'7']*`

  hex `"0x" ['0'-'9' 'a'-'f' 'A'-'F']*`

- *floating point constants* have the usual definition. They contain either a decimal point, or an exponent of the form `['e' 'E'] ('+' | '-')? decimal`.

- *string constants* are have the form `"s"`, where `s` is a sequence of characters, including escaped characters.

2.2 Grammar

The grammar of the language is as follows. The `typewriter` font is used for *terminals*, the *italic* font for *non-terminals*. The main complexity in the grammar is in declarations, which have lots of forms (wouldn't it be nice if it were so easy as "let x = e1 in e2"). A toplevel program (in a `.c` file) contains a sequence of variable, function, and type definitions.

prog:
  ϵ
  | *prog def*

def:
  var_defs
  | *fun_def*
  | *type_defs*

var_defs:
  type_specifier init_declarator_list `;`

fun_def:
  type_specifier declarator `{` *stmt_list* `}`

type_defs:
  `typedef` *type_specifier declarator_list* `;`

A type specifier is one of the builtin types, or a structure, or a type name that has been defined using `typedef`.

type_specifier:
`char` | `int` | `float`
| *struct_specifier*
| *typedef_name*

struct_specifier:
`struct` *identifier*$_{opt}$ `{` *struct_declaration_list* `}`
| `struct` *identifier*

struct_declaration_list:
struct_declaration
| *struct_declaration_list struct_declaration*

struct_declaration:
type_specifier declarator_list `;`

Now, here is where it gets more complicated. In a declaration, the type of the declaration is defined using stars, brackets, and parentheses in various incantations. How do you write the type of a function that returns a function? That's a good question. In any case, variable declarations can also contain initializers after an equals = sign. An *init_declarator* is a declaration with a possible initializer. A *declarator* doesn't allow the initializer.

init_declarator_list:
init_declarator
| *init_declarator_list* `,` *init_declarator*

init_declarator:
declarator
| *declarator* = *expr*

declarator_list:
declarator
| *declarator_list* `,` *declarator*

declarator:
pointer direct_declarator

direct_declarator:
identifier
| `(` *declarator* `)`
| *direct_declarator* `[` *expr* `]`
| *direct_declarator* `(` *parameter_list* `)`

pointer:
ϵ
| *pointer* *

Parameter lists add to the intrigue, because the parameter names do not have to be used in a declaration (although they may). We're going to force ANSI syntax in parameter declarations, not the classical form. That is, we accept the code.

```
int incr(int i) { return i + 1; }
```

We don't allow the following old-style code.

```
int incr(i)
int i;
{
   return i + 1;
}
```

Here we go. An *abstract_declarator* is a declarator with an optional name.

parameter_list:
parameter_declaration
| *parameter_list* , *parameter_declaration*

parameter_declaration:
type_specifier declarator
| *type_specifier abstract_declarator*$_{opt}$

abstract_declarator:
pointer
| *pointer direct_abstract_declarator*

direct_abstract_declarator:
(*abstract_declarator*)
| *direct_abstract_declarator*$_{opt}$ [*expr*$_{opt}$]
| *direct_abstract_declarator*$_{opt}$ (*parameter_list*)

Now we get to the "easy" part: statements and expressions. Statements are things that end with a semicolon. Expressions do not.

stmt:
;
| *expr* ;
| `if` (*expr*) *stmt*
| `if` (*expr*) *stmt* `else` *stmt*
| `while` (*expr*) *stmt*
| `for` (*expr* ; *expr* ; *expr*) *stmt*
| `return` *expr* ;
| `break;`
| { *stmt_list* }
def

stmt_list:
ϵ
| *stmt_list stmt*

expr:
char | *int* | *float* | *string* | *identifier*
| *preop expr*
| *expr postop*
| *expr binop expr*
| *expr* . *identifier*
| *expr binop* = *expr*
| (*expr*)
| *expr* [*expr*]
| *expr* (*expr_list*$_{opt}$)

expr_list:
    expr
    | *expr_list* , *expr*

preop:
    - | ! | * | & | ++ | --

postop:
    ++ | --

binop:
    + | - | * | / | %
    | << | >> | & | | | =
    | == | != | <= | >= | < | >
    | && | | |

Note that this grammar is highly ambiguous. You will need to add precedence declarations to get all the arithmetic, etc., to work out.

3 Checking out the code

Some of the code for this assignment will be provided for you. The distribution contains a definition of a type for the AST, another Set implementation (this time using red-black trees), a symbol table, and an interpreter.

We will be using CVS in this class to manage the distribution. CVS comes with most Linux distributions, and its on `mojave`. There are also versions for Windows out there, but then you should be shelling out the bucks and paying for Microsoft Cooperative Visual SourceSafe++ (MSCVS++) (that's a joke: you can get Windows versions of CVS from `www.cyclic.com`). CVS is pretty easy to use. The idea is this: we keep a central repository of the compiler source. When you want to build your compiler, you "check out" a copy of the compiler. You can work on your copy. When we make changes to the central copy, you can merge those changes into your code automatically.

This is a three-step process.

1. Login to the CVS server on `mojave` using the following command.

   ```
   % cvs -d :pserver:cs134@mojave.cs.caltech.edu:/cvsroot login
   ```

   It will prompt you for a password. The password is `cs134b`.

2. Go to the directory where you want to put the compiler, and check out your personal copy.

   ```
   % cvs -d :pserver:cs134@mojave.cs.caltech.edu:/cvsroot checkout fcc
   ```

   IF everything works, this will print out a lot of messages, and leave you with a directory called `fcc`, which contains subdirectories `mk`, `mllib`, and `fc_ast`. The `mk` directory contains some `Makefile` stuff, the `mllib` contains utilities like a `Set` implementation, and the `Symbol` module. The `fc_ast` directory is where you will do your work.

   You can build the compiler by running the `make` program in the `fcc` directory.

3. Later, if you want to merge any updates we have made to your copy of the compile, you would use the following command in the `fcc` directory.

```
% cvs update
```

You will get messages about the files you have changed. If you made changes to the same places that we did, you may get "conflicts." CVS will tell you where they are, and you will have to edit them by hand to correct the problem. The conflicts are always delimited by <<<< and >>>> brackets.

3.1 What's provided

3.2 Red_black_set and Red_black_table

These are new implementations of the Set and Table modules. You will rarely access them directly. Usually, you will use the SymbolSet and SymbolTable modules defined in symbol.mli.

3.3 Symbol

The Symbol module implements identifiers. Identifiers (for variables, type names, etc.) are ubiquitous in a compiler implementation, and it is useful to define some utility functions. We want an *abstract* representation of identifiers, so that their creation is tightly constrained (we don't want to introduce new identifiers by accident). The main signature for the Symbol module is as follows.

```
(*
 * Representation of symbols.
 *)
type symbol

(*
 * Create a new symbol with the given name.
 *)
val add : string -> symbol

(*
 * Make a new symbol.
 *)
val new_symbol : symbol -> symbol
val new_symbol_string : string -> symbol
```

The add function creates a new symbol from a string. The new_symbol function create *new* symbols that are guaranteed to be different from all the other symbols in the program. We'll use these functions a lot in later stages to make up unique variable names.

Also, we define the SymbolSet and SymbolTable modules here. We'll be using the SymbolTable a lot.

```
module SymbolSet : Set_sig.SetSig with type elt = symbol
module SymbolTable : Set_sig.TableSig with type elt = symbol
```

3.4 Fc_ast_type

This will be the main focal point of this Lab. Your goal is to reduce a program to an expression of type Fc_ast_type.expr. This is what an expression looks like.

```
type expr =
   IntExpr of int * pos
 | FloatExpr of float * pos
 | CharExpr of char * pos
 | StringExpr of string * pos
 | VarExpr of symbol * pos
 | AddrOfExpr of expr * pos
 | UArithExpr of uarithop * expr * pos
 | UnOpExpr of unop * expr * pos
 | BinOpExpr of binop * expr * expr * pos
 | BoolOpExpr of boolop * expr * expr * pos
 | SubscriptExpr of expr * expr * pos
 | ProjectExpr of expr * symbol * pos
 | ApplyExpr of expr * expr list * pos
 | AssignExpr of binop option * expr * expr * pos
 | IfExpr of expr * expr * expr option * pos
 | ForExpr of expr * expr * expr * expr * pos
 | WhileExpr of expr * expr * pos
 | SeqExpr of expr list * pos
 | ReturnExpr of expr * pos
 | BreakExpr of pos
 | VarDefs of var_init_decl list * pos
 | FunDecl of symbol * ty list * ty * pos
 | FunDef of symbol * var_decl list * ty * expr * pos
 | TypeDefs of var_decl list * pos
```

3.4.1 Position information

Each expression contains a "position" value, so that compile-time errors can be printed sensibly. The `pos` type is a 5-tuple.

```
(*
 * Position contains:
 *   1. filename
 *   2,3. starting line, starting char
 *   4,5. ending line, ending char
 *)
type pos = string * int * int * int * int
```

The position argument specifies the smallest line/character range containing the expression. The `Fc_ast_util` module defines some useful functions on positions. The `union_pos` function can be used to compute the characters spanned by two separate positions.

```
(*
 * Combine two positions.
 *)
val union_pos : pos -> pos -> pos

(*
 * Position functions.
 *)
val pos_of_expr : expr -> pos
val pos_of_type : ty -> pos
```

3.4.2 Basic expressions

The `Int`, `Char`, `String`, and `Float` expressions represent constants. A `VarExpr` represents a variable.

3.4.3 Unary operators

There are several forms of unary operations. A `UArithExpr` is unary arithmetic for the `++` and `--` operators. The `uarithop` is one of four cases:

```
type uarithop =
   PreIncrOp
 | PreDecrOp
 | PostIncrOp
 | PostDecrOp
```

The `UnOpExpr` is also a unary operation. The `unop` is one of the following cases.

```
type unop =
   UMinusOp             (* Arithmetic negation *)
 | UNotOp               (* Logical negation *)
 | UStarOp              (* Pointer dereference *)
```

3.5 Binary operators

There are three kinds of binary operations. Normal arithmetic uses the `BinOpExpr`. The `binop` is one of the following.

```
type binop =
   PlusOp               (* + *)
 | MinusOp              (* - *)
 | TimesOp              (* * *)
 | DivideOp             (* / *)
 | ModOp                (* % *)
 | BAndOp               (* & *)
 | BOrOp                (* | *)
 | BXorOp               (* ^ *)
 | LShiftOp             (* << *)
 | RShiftOp             (* >> *)
 | EqOp                 (* == *)
 | NotEqOp              (* != *)
 | LeOp                 (* <= *)
 | LtOp                 (* < *)
 | GeOp                 (* >= *)
 | GtOp                 (* > *)
```

C also allows these binary operations in an assignment statement, like `x += y`. Assignments use the `AssignExpr` form, which takes an optional binary operator.

The Boolean operations are handled separately with the `BoolOpExpr`. The definition of `boolop` is the following.

```
type boolop =
   LAndOp                (* && *)
 | LOrOp                 (* || *)
```

3.6 Projections

Array subscripting uses the `SubscriptExpr` form, which takes an expression for the array, and another for the subscript. Structure projection uses the `ProjectExpr` form, with an expression for the structure, and another for the label.

3.7 Control operations

The `SeqExpr ([e1; ...; en], pos)` represents a compound statement `{ e1; ...; en; }`.

The `IfExpr (e1, e2, Some e3, pos)` is a conditional expression eith an else statement.

```
if(e1) e2 else e3
```

If the third expression is `None`, the `if`-statement has no `else` case.

A `ForExpr(e1, e2, e3, e4, pos)` is a `for` loop.

```
for(e1; e2; e3)
   e4
```

The `WhileExpr(e1, e2, pos)` is a `while` loop.

```
while(e1)
   e2
```

The `BreakExpr` is a `break` statement.

3.7.1 Functions

Functions are called with the `ApplyExpr(f, args, pos)` form. The `ReturnExpr (e, pos)` is a `return e` statement.

3.7.2 Declarations and definitions

There are three kinds of declarations. "Variables" are declared with the `VarDefs` form, which list the variable name, it's type, and an optional initializer.

```
type var_init_decl = symbol * ty * expr option * pos
```

Functions are declared with a `FunDecl (f, params, ty, pos)`, where `ty` is the type of the values returned by the function, `params` are the type of the arguments, and `f` is the function name. If you like, you may use the `VarDefs` form instead for function declarations.

Functions are defined with the `FunDef (f, formals, ty, e, pos)` form. `ty` is the type of values returned by the function, `formals` is the formal parameter list, `e` is the function body, and `f` is the name of the function. The formal parameters are a list of variables and types.

```
type var_decl = symbol * ty * pos
```

Types are defined with the `TypeDefs` form, which specifies a list of type variable and their corresponding definitions.

3.7.3 Types

C has a simple type system. Here is the complete definition for types.

```
type ty =
   TypeChar of pos
 | TypeInt of pos
 | TypeFloat of pos
 | TypeId of symbol * pos
 | TypeArray of ty * expr option * pos
 | TypeStruct of ty_fields * pos
 | TypeFun of ty list * ty * pos
```

The `TyChar`, `TyInt`, and `TyFloat` correspond to the `char`, `int`, and `float` types. `TypeArray` is used for both pointers and arrays, a value of type `TypeArray (ty, e, pos)` is a pointer to a value of type `ty`. The `expr option` is an optional array bounds specifier (used mostly in initializers). For instance, in a declaration of the form `int a[10]`, the type of `a` would be

```
TypeArray (TypeInt pos1, IntExpr (10, pos2), pos3).
```

The `TypeId (v, pos)` form refers to a type defined with `typedef`, or one of the built-in types `char`, `int`, or `float`.

A `TypeStruct (fields, pos)` defines a `struct`. The `fields` is a list of labels and their types.

The `TypeFun (ty_args, ty_result, pos)` is the type of a function. The `ty_args` is a list of the types of the arguments (the list may be empty), and the `ty_result` is the type of values returned by the function.

3.8 Other files

There are sveeral more files you will find useful. The `Fc_ast_eval` module defines an interpreter for evaluating expressions. The `Fc_ast_util` defines some functions for printing expressions, and the `Fc_ast_exn` module defines some common exceptions.

It is not possible to make the parser totally functional. The `Fc_ast_state` module is used to keep track of some the the necessary state, including the current position in the file, and a set of symbols defined using `typedef`.

4 What you have to do

The distribution contains (very) incomplete versions of the lexer (`fc_ast_lex.mll`) and the parser (`fc_ast_parse.mly`). In this assignment, you will complete the implementation of these two files.

You should have *at most one* shift/reduce conflict (for the `if` statement), and you should have no reduce/reduce conflicts. Your grammar should be complete for the subset of C that we have defined.

Your compiler should correctly parse and evaluate the `test1.c` and `test2.c` files.

5 What to turn in

You should turn in your entire `fc_ast` directory in your `submit/lab2` directory on `mojave` (you should probably keep a copy in your CS account too). IF you are working in a group, only one of you should do the submission, and you should send mail to `cs134-admin` to tell us who the group is. If you had to modify anything in the `mllib` directory, include your changes.

In addition, you should include the following.

- A `README` file explaining what you did, how it works, and whether you had any problems.
- A `DIFF` file generated using the command `cvs diff` in the `fc_ast` directory. If you like, you can insert this into the `README` file, with brief explanations of what you changed.
- The files `test1.ast` and `test2.ast` generated using your compiler with the `-print_ast` option.

```
% ./fcc -print_ast test1.c > test1.ast
% ./fcc -print_ast test2.c > test2.ast
```

- Program output of the test runs. Use the following commands.

```
% ./fcc -eval test1.c -- test1 10 > test1.out
% ./fcc -eval test2.c -- test2 > test2.out
```