Homework 3

Due: 23 March 2012

Problem 3.1

On torvalds, copy the files in the directory `/courses/445/cparser-model` into your own. This contains a complete recursive-descent parser for the Project 3a subset, with a few bells and whistles that weren't required of your projects (for instance, keeping track of filenames and line numbers). Read and grok the code.

Particularly look at the parser functions themselves, in `cparser.cpp`, and modify them in the following way: in all the situations where a function's NULL return value is used to indicate "I can't handle the current input", modify its calling environment to first *check* if the peeked token is one that it could handle, so that a parse function is only called if it is definitely able to handle the (currently peeked token of the) input. Once that is complete, modify all the functions that return NULL to instead signal a syntax error using `syn_fail()`.

Problem 3.2

There are many places throughout the parser where `syn_fail()` is called to indicate unparsable input (including, if you've finished the previous problem, a few that you've added). Change all of these to provide an argument to `syn_fail` indicating what was *expected*—and be more specific than just the token type when possible. For instance, instead of

```
syn_fail("semicolon");
```

you might say

```
syn_fail("statement-ending semicolon");
```

instead.

Then, write short test files that each trigger a different syntax error. Each should include a comment that indicates what the error is, where it is, and why it's an error. E.g.

```
int x;
char ()
{}
// error on line 2: ident expected before lparen, missing function name
```

Terseness is fine; there will be a lot of these. (You probably want to put them in a subdirectory.)

Problem 3.3

Write a bison grammar for a slightly expanded version of our subset of C. It should accept the same language as the grammar below, although the `.yy` file does not need to be a line-for-line reproduction of this copy of the grammar. Non-terminals that weren't in the previous subset, and extra productions for existing NTs, are marked with a star.

Note that your grammar does not need to produce a parse tree, but should resolve the ambiguities in the below grammar in the same way as is done in C. (There are three nonterminals that have ambiguous parses according to this grammar, and they've all come up before.)

You should include test cases, both for files that parse and for files that shouldn't. (For this you can use many of the same files as in the previous problem.)

c-file:
- *top-level* EOF
- *top-level c-file*

top-level:
- *func-def*
- *func-decl* ⋆
- *var-decl*

func-def:
- *type ident* `(` *opt-params* `)` `{` *decls stmts* `}`

func-decl: ⋆
- *type ident* `(` *opt-params* `)` `;`

decls:
- *var-decl decls*
- ϵ

stmts:
- *stmt stmts*
- ϵ

var-decl:
- *type var-list* `;` ⋆

var-list: ⋆
- *var*
- *var-list* `,` *var*

var:
- *ident*
- `*` *var* ⋆
- *var* `[ ]` ⋆
- *var* `[` *int-literal* `]` ⋆

type:
- `char`
- `short` ⋆
- `int`
- `long` ⋆
- `float` ⋆
- `double` ⋆
- `void` ⋆

opt-params:
- *params*
- ϵ

params:
- *param-decl*
- *params* `,` *param-decl*
- *params* `,` `...` ⋆

param-decl:
- *type var* ⋆

compound-stmt:
- `{` *decls stmts* `}`

while-stmt:
- `while (` *expr* `)` *stmt*

stmt:
- `;` ⋆
- *expr* `;`
- *compound-stmt*
- *if-stmt*
- *while-stmt*
- `return ;` ⋆
- `return` *expr* `;`

if-stmt:
- `if (` *expr* `)` *stmt* ⋆
- `if (` *expr* `)` *stmt* `else` *stmt*

expr:
- *expr bin-op expr*
- *unary-op expr* ⋆
- *expr* `[` *expr* `]` ⋆
- `(` *expr* `)`
- *ident*
- *ident* `(` *opt-exprs* `)`
- *char-literal*
- *int-literal*
- *string-literal* ⋆

opt-exprs:
- *exprs*
- ϵ

exprs:
- *expr*
- *exprs* `,` *expr*

Binary operators include:

```
= + - * == != % / << >> < <= > >= && || += -= *= /=
```

Unary operators include:

```
+  ++  -  --  !
```