PS1 solutions

We're still working on the PS1 grades, but they should be out soon. The early data indicates most people did very well on the problem set, so it appears there is pretty solid understanding of the top-down parsing algorithms and implementation. Good work!

Part a:

Here is an equivalent grammar in LL(1) form. We left-factored the select and criteria productions and removed left-recursion from the col_list production. There was no ambiguity in the grammar.

```
S             ← SELECT cols FROM table opt_where $
opt_where     ← WHERE criteria | ∂
cols          ← * | col_list
col_list      ← col rest_of_list
rest_of_list  ← , col rest_of_list | ∂
col           ← id
table         ← id
criteria      ← int relop field | field relop field_or_int
field_or_int  ← field | int
relop         ← > | < | =
field         ← id
```

Here are the first sets:

```
First(field) = First(col) = First(table) = { id }
First(relop) = { < > = }
First(field_or_int) = First(criteria) = { id int }
First(col_list) = { id }
First(rest_of_list) = { , ∂ }
First(cols) = { * id }
First(opt_where) = { WHERE ∂ }
First(S) = { SELECT }
```

Here are the follow sets:

```
Follow(S) = Follow(opt_where) = Follow(criteria) =  Follow(field_or_int) = { $ }
Folllow(cols) = Folllow(col_list) =  Follow(rest_of_list) = { FROM }
Follow(col) = { , FROM }
Follow(table) = { WHERE $ }
Follow(field) = { < > = $ }
Folllow(relop) = { id int }
```

Here's a sample function from our recursive descent parser to show the general implementation:

```
static void ParseOptionalWhere()
{
     switch (lookahead) {
         case T_WHERE:    // Predict opt_where -> WHERE criteria
            MatchToken(T_WHERE, "WHERE");
            ParseCriteria();
            break;
         case T_DOLLAR:   // Predict opt_where -> epsilon
```

```
            break;
         default:
            ReportParseFailure("WHERE or $", yytext);
      }
   }
```

Part b:
Here is an equivalent grammar in LL(1) form. We removed the ambiguity by enforcing the precedence as stated and chose left-associativity for the binary operators. (It was also okay to associate right, this will avoid the left recursion, but it then requires left-factoring. If it's not one thing, it's another..)

B ← T B' $
B' ← or T B' | ∂
T ← F T'
T' ← and F T' | ∂
F ← not F | true | false | id | (B)

The nullable non-terminals are { B' T' }

Here are the first sets:

First(B) = Field(T) = First(F) = { true false id not (}
First(B') = { or ∂ }
First(T') = { and ∂ }

Here are the follow sets:

Follow(B) = Follow(B') = { $) }
Folllow(T) = Follow(T') = { $) or }
Folllow(F) = { $) or and }

Here is the completed table for a top-down predictive parser:

Top of stack	true	false	id	not	or	and	(	)	$
B	B ← T B'	B ← T B'	B ← T B'	B ← T B'			B ← T B'		
B'					B' ← or T B'			B' ← ∂	B' ← ∂
T	T ← F T'	T ← F T'	T ← F T'	T ← F T'			T ← F T'		
T'					T' ← ∂	T' ← and F T'		T' ← ∂	T' ← ∂
F	F ← true	F ← false	F ← id	F ← not F			F ← (B)		

Here is trace on the input false or (not fun)

PARSE STACK	REMAINING INPUT	PARSER ACTION
B$	false or (not fun)$	Predict B ← T B'
TB'$	false or (not fun)$	Predict T ← F T'
FT'B'$	false or (not fun)$	Predict F ← false

falseT'B'$	false or (not fun)$	Match false
T'B'$	or (not fun)$	Predict T' → ε
B'$	or (not fun)$	Predict B' → or TB'
orTB'$	or (not fun)$	Match or
TB'$	(not fun)$	Predict T → FT'
FT'B'$	(not fun)$	Predict F → (B)
(B)T'B'$	(not fun)$	Match (
B)T'B'$	not fun)$	Predict B → FB'
TB')T'B'$	not fun)$	Predict T → FT'
FT'B')T'B'$	not fun)$	Predict F → not F
notFT'B')T'B'$	not fun)$	Match not
FT'B')T'B'$	fun)$	Predict F → id
idT'B')T'B'$	fun)$	Match fun (id)
T'B')T'B'$	)$	Predict T' → ε
B')T'B'$	)$	Predict B' → ε
)T'B'$	)$	Match)
T'B'$	$	Predict T' → ε
B'$		Predict B' → ε
$	$	Match $, success!