Homework #8

Question 1 (6 pt.)

Consider the grammar for the version of the MiniC-to-LLVM compiler presented in class with support for variable declarations and assignments (only), using the baseline implementation provided in the additional material. Extend the grammar with the following two rules, where *S* stands for *statement*, as *E* stands for *expression*:

$$S \rightarrow * \; E \; = \; E \; ;$$

$$E \rightarrow * \; E$$

The first production extends assignments with support to the dereference operator, used for writing purposes. The second production extends expressions to support the same operator for reading purposes. Notice that these two cases have different semantics, and should produce different code. Extend the parser to generate correct LLVM code for the dereference operator in its two forms. Upload your code in a file named `q1.y`.

Your parser should behave properly in the following scenarios:

Case 1

```
void f()
{
    int *x;
    int y;

    y = *x;
}
```

The parser produces LLVM code to load the value pointed to by `x`.

Case 2

```
void f()
{
    int x;
    int y;

    y = *x;
}
```

The parser reports an error, noting that `x` is not a valid pointer.

Case 3

```
void f()
{
    int *x;
    int y;

    *x = y;
}
```

The parser produces LLVM code to store a value in the memory address pointed to by `x`.

Case 4

```
void f()
{
    int x;
    int y;

    *x = y;
}
```

The parser reports an error, noting that `x` is not a valid pointer.

Question 2 (4 pt.)

Extend the same MiniC-to-LLVM parser to support the unary "–" operator, used as a prefix to invert the sign of the expression on its right. Upload your code in a file named `q2.y`. Your parser should accept the following input, and emit the proper LLVM code for it:

```
void f()
{
    int x;
    int y;
    y = -x;
}
```