Notes for Programming in C Lab Session #5

15 October 2018

1 Introduction

The purpose of this lab session is to practice debugging an existing program, using the ASan and UBSan tools built in to GCC and clang.

2 Overview

Once built, the `lab5` executable will read the arithmetic expressions passed to it as a command-line argument. Then, the program will parse the argument into a parse tree, evaluate the parsed expression, and then print both the parse tree and the result of the square of the evaluation.

```
$ ./lab5 "3"
3 squared = 9

$ ./lab5 "3+4"
(3+4) squared = 49

$ ./lab5 "3+4*5"
(3+(4*5)) squared = 529

$ ./lab5 "3+4*5+2"
((3+(4*5))+2) squared = 625

$ ./lab5 "3+4*(5+2)"
(3+(4*(5+2))) squared = 961

$ ./lab5 "(3+4)*(5+2)"
((3+4)*(5+2)) squared = 2401
```

The terms of the syntax this calculator accepts are:

- positive integer literals, such as `12` or `3124`.
- The sum of two terms, such as `2+3` or `(2*3)+4`.
- The product of two terms, such as `2*3` or `1*2*3`.
- A parenthesized term, such as `(1)` or `(2*3)` or `(((2*3+1)))`.
- Addition and multiplication associate to the left – i.e., `1+2+3` is the same as `(1+2)+3`.
- Addition is lower precedence than multiplication – i.e., `1+2*3` is the same as `1+(2*3)`.

For simplicity, no whitespace is permitted in arithmetic expressions, and neither is subtraction:

```
$ ./lab5 "1 + 2"
parse error

$ ./lab5 "1-2"
parse error
```

However, this is the theory! This program has been carefully salted with a few bugs, and it will crash on most inputs. Your task is to find and fix the bugs in this program. Hopefully, the use of UBSan and ASan will make it much easier to find these bugs than before!

3 Instructions

1. Download the `lab5.tar.gz` file from the class website.
2. Extract the file using the command `tar xvzf lab5.tar.gz`.
3. This will extract the `lab5/` directory. Change into this directory using the `cd lab5/` command.
4. In this directory, there will be files `lab5.c`, `expr.h`, `expr.c`, `parse.h`, and `parse.c`.
5. There will also be a file `Makefile`, which is a build script which can be invoked by running the command `make` (without any arguments). It will automatically invoke the compiler and build the `lab5` executable.
6. Once built, this file accepts command-line arguments to evaluate arithmetic expressions and square them.
7. To invoke the compiler with debugging instrumentation turned on, invoke make with the command `make sane`, which will turn on the address sanitizer and undefined behaviour sanitizers.

4 Documentation of the Types and Functions

4.1 The `expr.h` module

- The expression data type:

```c
typedef enum type {LIT, PLUS, TIMES} expr_type;
typedef struct expr * expr_t;
struct expr {
  expr_type type;
  union {
    int literal;
    struct pair {
      expr_t fst;
      expr_t snd;
    } args;
  } data;
};
```

  The `expr_t` type represents syntax trees of arithmetic expressions. It is a pointer to a struct, whose `type` field is an enumeration saying whether this expression is a literal `LIT`, an addition node `PLUS`, or a multiplication node `TIMES`. If the `type` field is `LIT`, the `data` field will be the `literal` branch of the union, storing the literal integer this node represents. If `type` field is `PLUS` or `TIMES`, the `data` field will be in the `pair` branch of the union, with the `fst` and `snd` representing the left- and right-hand sides of the arithmetic operation.

- `expr_t mkLit(int n);`

  Construct a fresh `expr_t` representing the literal `n`.

- `expr_t mkPlus(expr_t e1, expr_t e2);`

  Construct a fresh `expr_t` representing the sum of `e1` and `e2`.

- `expr_t mkTimes(expr_t e1, expr_t e2);`

  Construct a fresh `expr_t` representing the product of `e1` and `e2`.

- `int eval_expr(expr_t e);`

  Return the integer which is the result of evaluating the expression `e`.

- `void print_expr(expr_t e);`

  Print the expression `e` to standard output.

- `void free_expr(expr_t e);`

  Free the memory associated with the expression `e`.

- `expr_t copy(expr_t e);`

  Construct a fresh copy of the expression tree `e`.

4.2 The `parse.h` module

- `int parse_int(char *s, int i, expr_t *result);`

  Parse an integer expression from the string `s`, beginning at position `i`. If the parse is successful, this function returns an integer index to the first character after the matched string, and writes the parse tree to the `result` pointer.

- `int parse_atom(char *s, int i, expr_t *result);`

  Parse an *atom* (i.e., either an integer or parenthesized expression) from the string `s`, beginning at position `i`. If the parse is successful, this function returns an integer index to the first character after the matched string, and writes the parse tree to the `result` pointer.

- `int parse_term(char *s, int i, expr_t *result);`

  Parse a term (i.e., a product of atoms, such as `1 * (2+3) * 4`) from the string `s`, beginning at position `i`. If the parse is successful, this function returns an integer index to the first character after the matched string, and writes the parse tree to the `result` pointer.

- `int parse_expr(char *s, int i, expr_t *result);`

  Parse an expression (i.e., a sum of terms, such as `1 + 2*3 + (4*(5+6))`) of multiplied expressions from the string `s`, beginning at position `i`. If the parse is successful, this function returns an integer index to the first character after the matched string, and writes the parse tree to the `result` pointer.

- `int parse(char *s, int i, expr_t *result);`

  Parse an expression as with `parse_expr`, but return NULL if the parse doesn't consume the whole string.