Nipping Bugs in the Bud – Eliminating Student Misconceptions in Introductory Computer Science Classes

Nivedita Chopra, advised by Roy Maxion and Robert Simmons

Background

Bugs are a major problem in software development, and bugs in software that is widely used can have far-reaching consequences, as was recently seen in the case of the Heartbleed bug in OpenSSL. While slips in syntax can be detected and fixed easily, oftentimes bugs are caused by misconceptions in the programmer's thought process. Such misconceptions ought to be detected and subsequently remedied early on, preferably in introductory and intermediate programming classes, so as to ensure that the programmers of tomorrow are much less likely to commit certain kinds of errors.

Aim

Our aim is to catalog the bugs committed by students in the Principles of Imperative Computation course (15-122), which is taught in C0 and C. By analyzing these bugs, we hope to gain an insight into the misconceptions behind them. Based on our analysis, we will propose improved teaching methods to eliminate these misconceptions.

Data

Bugs were obtained and recorded in three ways :
1) Using a Google form filled in by students.
2) Through observation of student code during lab and office hours.
3) Through analysis of posts on Piazza, an online Q&A forum for the class.

Methods

Each week, we analyzed the bugs seen in the previous week, and attempted to classify them using the IEEE Standard Classification for Software Anomalies (2010), as a first step towards developing our own taxonomy of bugs observed in introductory computer science classes. For the common bugs seen each week, we tried to determine the misconceptions that led to the bug. We then proposed some changes to the course for the Spring 2015 semester, that are anticipated to eliminate or mitigate these misconceptions

CATEGORIZATION OF BUGS OBSERVED IN 15-122

Logic Bug (as per the IEEE classification)

Traverse through a linked list performing a certain operation on each element

```
while (L->next != NULL) {
  //do something;
  L = L->next;
}
```

Bug : *Misses the last element in the list*

Data Bug (as per the IEEE classification)

Write a function to check if a given integer is in a linked list

```
bool is_in(list L, int n) {
  while(L->start != NULL) {
    if(L->start->data == n) return true;
    L->start = L->start->next;
  }
  return false;
}
```

Bug : *Destroys the linked list during an effect-free operation*

Interface Bug (as per the IEEE classification)

Write a function that removes the green component of a given pixel

Interface :

```
pixel make_pixel(int a, int r, int g, int b);
int get_green(pixel p);
```

Implementation :

```
typedef int pixel;
```

Client code (written by student) :

```
pixel remove_green(pixel p) {
  return (p & 0xFF00);
}
```

Bug : *Relies on the implementation while writing client code, which should only deal with the interface*

Comprehension Bug (not seen in the IEEE classification)

1) Create the list containing 1, 2, and 4, in order, using the cons and nil functions
[list cons(int, list); list nil();]

```
list L = cons(1, cons(2, cons(4)));
```

Bug : *Fails to realize that the cons function takes two arguments*

2) The offset is a signed 16 bit integer that is given as a two-byte operand to the instruction. (Instructions are stored in an array of (unsigned) bytes (ubyte *P))

```
int16_t o1 = (int16_t)(int8_t)P[pc+1];
int16_t o2 = (int16_t)(int8_t)P[pc+2];
int16_t offset = ((o1 << 8) | o2);
```

Bug : *Casts both o1 and o2 into signed integers to make resultant offset signed. Sign extension on o2 may alter the final quantity.*

Results

The bugs observed were logic bugs (60 instances), data bugs (21), and interface bugs (7), as per the IEEE Standard Classification for Software Anomalies (2010). An additional category of bugs, with 32 observed instances, emerged here that was not specified in the IEEE classification. These can be called "comprehension errors." Often, when a task is described to students, they are unsure about which paradigm or algorithm to use to accomplish it. Students are often unfamiliar with reading specifications, and are hence likely to misunderstand them and to make incorrect assumptions while programming. Due to having limited prior programming experience, and possibly because they are using new tools, students are often stumped by error messages and warnings, both from the compiler and from Autolab, which auto-grades submitted code.

Further Work

We noticed many bugs arising due to lack of attention given to edge cases. We feel that encouraging students to list edge cases before coding may enable them to write more correct code, with fewer bugs due to edge cases.

Based on the fact that many students committed the same bugs, and that similar types of bugs were seen throughout the semester, we feel that maintaining a record of one's bugs might help debug better. We are planning to introduce the concept of a "Bug Diary" as an optional, and highly recommended, experiment in 15-122 for Spring 2015.

Impact

We anticipate that the results of our research will help eliminate many student misconceptions. This is likely to improve the quality of code written by students in further classes, and in their future careers.

References

"IEEE Standard Classification for Software Anomalies," IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), Jan. 7 2010.
Beizer, Boris. Software Testing Techniques. 2nd ed. 1990.
Saj-Nicole Joni, Elliot Soloway, Robert Goldman, and Kate Ehrlich. 1983. Just so stories: how the program got that bug. SIGCUE Outlook 17, 4 (September 1983), 13-26.