Spez. Kapitel aus Softwareentwicklung
Formal Specification and Development of Software
Johannes Kepler Universität, Linz, Austria, 2023

Project

Weighting
50%

Due in
2023-06-09 23:59UTC+01 (Friday evening)
by email to dlightfoot@brookes.ac.uk.

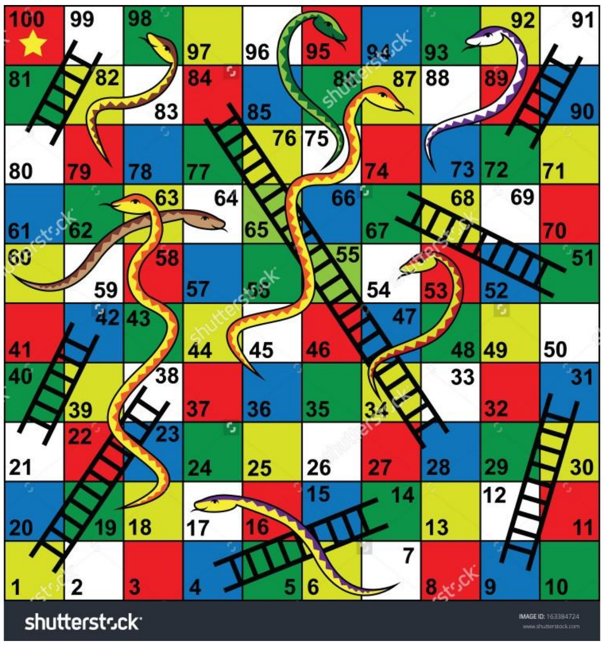
First word of title must be 'Linz'.
Attach a *Microsoft Word* (or *pdf*) document (or *zipped* equivalent).
You may send a scanned, handwritten document but I shall not be able to give marks if I cannot read it.

What you have to do
Put the *title*, *your name* and *your student number* on the first page of your document.
Give brief explanations of any assumptions you make. If you have any questions about what is required please send email to David Lightfoot at the address above.

Make use of the *Zed* font. On Windows, copy the file *ZED____.TTF* into the folder Windows/Fonts. In Microsoft Word use *Insert Symbol* to put the characters into your document and save your file with the option 'Embed Truetype fonts'.

Answer *both* questions. This must be *individual* work, not group work.

Question 1 – formal specification of the game of Snakes and Ladders



http://www.tendirham.com/298-large_default/snakes-and-ladders-board-game.jpg

Each player starts with a token on the starting square (usually the "1" grid square in the bottom left corner, or simply, off the board next to the "1" grid square). Players take turns rolling a single die to move their token by the number of squares indicated by the die roll. Tokens follow a fixed route marked on the game-board which usually follows a *boustrophedon* (ox-plough) track from the bottom to the top of the playing area, passing once through every square. If, on completion of a move, a player's token lands on the lower-numbered end of a "ladder", the player moves the token up to the ladder's higher-numbered square. If the player lands on the higher-numbered square of a "snake" (or chute), the token must be moved down to the snake's lower-numbered square.
The player who is first to bring their token to the last square is the winner.
https://en.wikipedia.org/wiki/Snakes_and_Ladders

a) Explain what aspects of the game as described above are aspects of the *view* (how it *looks*) rather than the underlying *abstract model* (how it *is*).
(For example, does the game *have to be* played on a 10 by 10 square grid, as in the picture? Does it matter *how many players* there are?)
Include any further constraints you perceive on the board and positions of snakes and ladders that are not mentioned in the above description and include them in your formal specification.

In the remainder of this work consider only the *abstract model*.
Include brief, narrative, English explanations of each of your schemas.

5 marks

b) Write names and descriptions for the *basic types* that needed for a Z specification of the game.

2 marks

c) Devise a Z schema to describe the current state for each player and of the snakes and the ladders. Include all *invariant* properties (constraints).

5 marks

d) Write a schema for an *initial* starting state and show informally that this state satisfies the invariant properties.

5 marks

e) Write an operation schema to make a *move* for a given *player* and a given number on the *die*. Explain informally how your schema maintains the invariant properties of the state.

5 marks

f) Write an enquiry schema that returns the player who has won, if there is one yet.

3 marks
Total 25 marks

Question 2 – formal program derivation of Zune code

Read about the" Zune Bricking code" on Guardian web page:

http://www.theguardian.com/technology/blog/2009/jan/01/zune-firmware-mistake

and elsewhere. See references list at end of this document.

Note that the English expression "The Zune's real-time clock stores the time in terms of *days and seconds since January 1st, 1980*", in the Guardian report is ambiguous, where the days are concerned. For example: How many days since January 1st, 1980, is January 1st 1980?

This ambiguity can be resolved by the fact that the software was noticed to go wrong just after midnight at the start of December 31, 2008.

The original code takes the number of days from the start of the origin year and works out the corresponding year and day number in that year.

Here it is: (in C)

```
year = ORIGINYEAR; /* = 1980 */
while (days > 365) {
    if (IsLeapYear(year)) {
            if (days > 366) {
                    days -= 366; year += 1;
            }
    } else {
            days -= 365; year += 1;
    }
}
```

a) **Assuming** the existence and availability of correct *Spec#* methods with specifications:
static bool **IsLeapYear**(int y)
requires y >= 1980;
ensures result == "year y is a leap year";

and
static int **DaysInMonth**(int y, int m)
requires y >= 1980 && 1 <= m && m <= 12;
ensures result == "number of days in month m of year y";

using *DaysInMonth*, **write** the ***implementation*** for the method with this specification in *Spec#*:
static bool **IsValidDate**(int y, int m, int d)
requires y >= 1980;
result == "y, m, d is a valid date";
*{**implementation***
}

3 marks

b) Complete the Spec# ***specification*** (not *implementation*) of a method with heading:
static int **DaysSince1Jan1980**(int year, int month, int day)
requires …;
ensures …;

that need only work where *y*, *m*, *d* constitute a *date* no earlier than the start of 1980. You may create your own auxiliary functions if you wish (for example *DaysInYear*).

3 marks

c) Following the methods taught in the formal-derivation part of the module, *derive* an implementation of the method *DaysSince1Jan1980* and annotate it with suitable loop *invariants* to show its partial correctness.

Hint: You will find it easiest to implement this in two parts: firstly summing the days in the whole years since 1980 and then the days since the start of year *y*.

Big hint: do **not** try to do this by writing the program first and then trying to prove it correct; that is **much more difficult** than deriving the implementation hand-in-hand with showing its correctness.

You may use material from the module's formal-derivation lectures without acknowledgement.

Requirement: you must not use *break*, *continue*, *return*, *goto* or any other "*crypto-goto*" statements in any of your implementations in this work! They are not necessary and they defy the ideas of structured programming and make formal proof of your implementation very difficult.

4 marks

d) Identify a suitable *bound function* for each loop in your method and explain your choice. Augment the annotation of your implementation to show its *total* correctness.

4 marks

e) Derive a *Spec#* implementation to be the inverse operation to *DaysSince1Jan1980*:

void **DaysBackToDate**(int days; out int y, out int m, out int d)
requires days > 0;
ensures IsValidDate(y, m, d) && days == DaysSince1Jan1980(y, m, d);

including invariants that show it to be *partially* correct.

Note: an *out* parameter in C# is like a *var* parameter in Pascal. It requires the actual parameter to be a variable and it carries a value out of the method through the parameter.

5 marks

f) Identify suitable bound functions and augment the annotation of your implementation to show its total correctness.

2 marks

g) Study the some of the many posts about the Zune 'bricking code'. Criticise them where you think they are wrong or poorly expressed or where the 'corrections' offered are either wrong or messy. Devise your own explanation of what is wrong with the Zune code, making reference to the ideas of formal derivation.

4 marks
Total 25 marks

Dijkstra guarded-command notation
You might find it useful to develop your implementation of the part that finds the *day in year* from *days* by using Dijkstra's guarded-command notation; it offers a very simple, elegant solution for this task.

Steps
- specify in form of pre- and post-condition;
- find suitable loop invariant;
- determine suitable guard;
- determine initialisation that establishes the invariant;
- determine body that maintains invariant;
- show that invariant conjoined with negation of guard implies post-condition;
- find bound (variant);
- use bound to prove termination.

Template for a loop

pre
initialisation
invariant

```
while guard do
  $invariant \land guard \land bound = B \land B > 0$
  body
  $invariant \land bound < B$
end
$invariant \land \neg guard \Rightarrow post$
```

References

http://www.theguardian.com/technology/blog/2009/jan/01/zune-firmware-mistake

http://bit-player.org/2009/the-zune-bug

http://en.wikipedia.org/wiki/Zune_30

http://bits.blogs.nytimes.com/2008/12/31/the-day-microsoft-zunes-stood-still/?hp&_r=0

http://latimesblogs.latimes.com/technology/2008/12/zune-30-shutdow.html