Questlang: a language for verifying quest specifications

A project for CS 421: Programming Languages and Compilers

by *Octavian-Mircea Sebe* and *Dominic Jones*

Overview

In interactive fiction and computer games, it is common to have content structured around the idea of a quest. A quest is an objective given to a player, usually a set sequence of commands that must be fulfilled. A quest may be modelled as a context free grammar, per Doran & Parberry 2011. Trivial tasks (e.g. travel, pick up, or talk) are represented by terminals and non-trivial, or compound, tasks are represented by nonterminal (e.g. a heist). Each of these terminals and non-terminals may have arbitrary data associated with them, essentially variables which bind to objects in the world.

This is an effective model for the generation of quests, useful for automatically creating new content on-the-fly for players without author intervention. However, this approach may also be used to validate human-authored quests. A quest, and its actions, should adhere to some internal logic of a world. If it does not, this can cause bugs, resulting in quests that are impossible to complete, ruining a player's experience. If a player cannot complete a quest, a designer would ideally desire feedback as to why. We can think of the quest as a programme written in this quest language, with the game environment corresponding to a programme environment and the quest tree modelled recovered from a lexing and parsing task. Finally, given clear semantics an execution model may be defined which will either execute successfully (e.g. the quest is completable) or fail, indicating why.

We proposed to implement this language to answer the question above: given a description of a game environment and a quest in the language, is this quest completable? If not, what prevents the quest from being valid? We describe our language below.

Note an important point of difference between our language and Doran & Parberry 2011: due to this language being intended for validation of human-authored quests, we jettison the motivational nonterminals and introduce author-defined subquests to sensibly group atomic actions.

Implementation

The project consists of 2 separate parts: the **Questlang** lexer/parser and the quest validator (performing semantic evaluation on an Abstract Syntax Tree). Each of these 2 parts can be swapped for another piece of code, without affecting the other part. The only point where the 2 parts join together is in the definition of the Abstract Syntax Tree (in `abstract_syntax_tree.ml`).

Other major components include `lexer.mll` and `parser.mly`: the lexer and parser respectively. `semantics.ml` implements the validator itself, `validate.ml` contains some code to further parse the AST into a format that finally is processed. `main.ml` is the entry point into the programme. See the repository structure section for more information.

We have implemented the main goal: that of a quest validator. It contains a small but flexible set of atomic quest actions: goto, get, kill, use, and require, which we believe covers the vast majority of actions in interactive fiction (e.g. RPG) games. The language also includes logical predicates to be used with the `require` action (a simple form of polymorphismm because it also takes items), and as mentioned above, user-defined parametrised subquests which may be used and reused in a quest.

We did not implement the motivation nonterminals (e.g. wealth, protection) mentioned in the paper, because we found that such a structure was more useful for generation than validation. We pared back the set of atomic actions to something we believe reflects most of the content in games and that authors would find useful. Differing from the paper, we did implement logic and user-defined subquests. See the goals section for more information.

The workload for this project was mainly split in two with Mircea Sebe working on the quest validator and Dominic Jones working on the lexer and parser, and both of us working on documentation, tests, Makefile, reports and code review. After agreeing on the structure of the AST we were both able to do most of the work asynchronously and independently.

The AST went through 3 iterations before converging on its current form, and after integrating lessons from the class. One such lesson was why it would be better to have the Actions as part of their own `unaryAction` type which is combined in a tuple with its argument (e.g. `Kill * Wolf`), rather than have an overarching Actions type with parametric constructors for each action.

An interesting task was also deciding what each action should do. In the end we went with what seemed most sensible and interesting to us.

The compiling and testing frameworks are encapsulated in the project's `Makefile`. The repository consists of a set of `.ml` files that comprise a library (`LIB`). In order to generate the main project executable, `questlang`, the library files are compiled using `ocamlopt` together with `main.ml` which is the entry point of our application.

Testing

In order to do unit testing, we can create another `.ml` file (`semantics_tester.ml`) that is compiled and run against the same library. In the case of `semantics_tester.ml`, we define some ASTs by hand and check the validation message that we get after validation with the expected message. See the comments in the respective file for more information on that.

In order to do integration testing, we will evaluate quests (`TEST_FILES` in the Makefile) written directly in Questlang and compare the output of this validation to the associated `.out.golden` file (using `diff`).

Both unit and integration testing are performed by running `make test` which will end in a return code of 0 and print no red error messages when everything works fine.

Our tests are structured to test each feature of the language we implement. For example,

- `quest-example.ql`: tests basic world construction, subquest definition, built-in functions, parameters and arguments, the atomic actions themselves, and logic.
- `quest-example-2.ql`: further tests the subquest system.
- `quest-example-3.ql`: further tests the subquest system, including subquests referencing other subquests. Tests multiple world and main quest definitions in one file.

Repository Structure

All source files, test files, executables and infrastructure files reside in the top level directory. Given the scale of this project, we didn't find this to be a problem but if the project continues to grow, we will consider separating the source, test files and generated executables into their own subdirectories.

The Questlang lexer is defined in `lexer.mll`.

The Questlang parser is defined in `parser.mly`.

All the validation of an already built AST happens in `semantics.ml`.

`utils.ml` and `validate.ml` just provide us with some useful functions.

Project Goals

The project is currently functional and able to take an arbitrary Questlang file to a validation report. There currently are no known bugs in the code.

The project proposal was open ended to the extent to which we would implement the original paper that this project is based off of, with the two of us aiming towards implementing as much of it as we could in the alloted time.

While we didn't implement many of the basic Actions specified in the original paper, we successfully implemented a very expressive subset of it, encapsulating what amounts to function calls, local variable bindings, complex logical expressions and more. We believe that expanding upon the semantics of our current Actions and adding new Actions would be a routine task given the framework that we currently have in place.

One shortcoming of the project is the lack of a way for the user to specify what "using" an item should do. At the moment, "using" an item will just check that the player has it and then discard it. But this basic Action can be used in tandem with other Actions and subquests to achieve more expressive semantics. For instance, a subquest could be defined as follows:

```
Subquest UseTeleportPotion ()
  use TeleportPotion
  goto MountainSummit
```

and running this quest will simulate the use of a teleportation potion.

In conclusion, we believe that the project shines in creating a framework that is easy to extend, easy to use, and very informative to a user looking to develop a quest for a video game or just to have fun.

Usage

To use this tool:

1. Run `make`
2. (optional) Run `make test` to run the unit and integration tests
3. Run `./questlang my-quest.ql` to validate a quest written in questlang, stored in `my-quest.ql`

See `quest-example.ql` for a sample questlang file.

To clean up the project directory run `make clean`

Actions

- `goto` - Changes the player's current location
- `get` - Tries to collect an item and add it to the player's inventory if it is at the player's location
- `kill` - Kills a monster if it is at the player's location
- `use` - Uses up one of the player's held items
- `require` - Checks if a certain predicate about the world is true

Code listing

As a PDF (or HTML document) is not the most appropriate format for viewing code, we reproduce select samples of code here. All of the files in the codebase are commented, which should suffice as a code listing, and we suggest that you peruse them in this order:

1. `abstract_syntax_tree.ml`: a definition of our AST in OCaml types.
2. `lexer.mll`: The lexer.
3. `parser.mly`: The parser.
4. `semantics.ml`: The core of the quest validator.
5. `semantics_tester.ml`: Testing just for the semantics.
6. `validate.ml`: Some helper code belonging after parsing but before evaluation.
7. `utils.ml` and `main.ml`: Utilities and our main entry point.

Definition of the AST from `abstract_syntax_tree.ml`:

```
type var = string;;
type locationId = LocationLiteral of string | NullLocation;;
type characterId = NPCLiteral of string | PlayerC;;
type itemId = string;;
type subquestId = string;;

(* Predicates are statements about the state of the world.
   They evaluate to booleans at a particular world state *)
type predicate =
    | HeldPred of itemId
    | DeadPred of characterId
    | AlivePred of characterId
    | AtPred of characterId * locationId;;

(* Conditions take the idea of predicates further by adding logical
connectives to them *)
type condition =
    | CondAnd of condition * condition
    | CondOr of condition * condition
    | CondImplies of condition * condition
    | CondNot of condition
```

```
    | CondPred of predicate;;

(* The type of parameter expressions that will be evaluated when
encountered while running the quest *)
type paramExp =
    | VarExp of var
    | LocationExp of locationId
    | ItemExp of itemId
    | CharExp of characterId
    (* Builtin function for getting the location of another parameter *)
    | GetLoc of paramExp
    | CondExp of condition;;

(* A world entry describing something about the world.
   A list of these is used to construct the initial world state *)
type worldEntry =
    | CharWorldEntry of characterId * locationId
    | ItemWorldEntry of itemId * locationId
    | LocationWorldEntry of locationId
    | VulnerabilityWorldEntry of characterId * (itemId list);;

(* Actions that the player can take *)
type unaryAction =
    | Require
    | Goto
    | Get
    | Kill
    | Use;;

(* One atomic component of a Quest: It can be either an action,
   a variable binding or a call to an external subquest*)
type questExp =
    | ActionExp of unaryAction * paramExp
    | LetExp of var * paramExp
    | RunSubquestExp of subquestId * (paramExp list);;

(* The type of subquests. Subquests can be tought of as functions
   that are called from the body of a quest with a list of arguments
   to be substituted for the formal arguments when running the subquest *)
type subquestEntry = subquestId * ((var list) * (questExp list));;

(* The main AST that we get from parsing, bundling together all that we've
seen earlier *)
type _AST = {
    world : worldEntry list;
    subquests : subquestEntry list;
    (* We may have multiple quests that are each validated individually
       against the same starting world state *)
    mainQuests : (questExp list) list;
};;
```

Example of one pattern matching branch of our main quest evaluator in `semantics.ml`:

```
| (Kill, (CharRes c)) -> (match c with
                | PlayerC -> Left (stepNo, "The player character cannot
kill themselves")
                | npc -> (match lookupPlayerLoc ws with
                    | None -> Left (stepNo, "Player is at an invalid
location")
                    | Some (_, npcs) -> (match extract npcs npc with
                        | None -> Left (stepNo, "NPC does not exist at
player's location")
                        | Some npcs' -> (match mapLookup ws.vulnerability
npc with
                            | None -> Left (stepNo, "NPC is invincible")
                            | Some vItems -> if exists (fun x -> mem x
ws.player.inventory) vItems
                                then recurse
                                    { (unsafeSetNpcsAtPlayerLoc ws npcs')
with
                                    charsDead = npc :: ws.charsDead;
                                    charsAlive = filter (fun x -> x <>
npc) ws.charsAlive
                                    }
                                else Left (stepNo, "Player cannot kill the
NPC")
                            )
                        )
                    )
                )
```

Example of a function used to populate the runtime World State from the data obtained via parsing, from `semantics.ml`:

```
let rec populateWorldState worldData world = match worldData with
    | [] -> Right world
    | worldE :: worldData' -> let recurse = populateWorldState worldData'
in ( match worldE with
        | CharWorldEntry (chr, loc) -> (match chr with
            | PlayerC -> (match world.player.location with
                | NullLocation -> recurse { world with player = {
world.player with location = loc }}
                | _ -> Left "Error: Player's starting location was set
twice")
            | npc -> if mem npc world.charsAlive then Left "Error: NPC's
location was set twice" else
                recurse { world with
                    charsAlive = npc :: world.charsAlive ;
                    worldMap = mapUpdate world.worldMap loc (fun (items,
npcs) -> (items, (npc :: npcs))) ([], [ npc ])
                }
            )
        | ItemWorldEntry (itm, loc) -> if mem itm world.allItems
```

```
            then Left "Error: item's location was set twice"
            else recurse { world with
                worldMap = mapUpdate world.worldMap loc (fun (items, npcs)
-> ((itm :: items), npcs)) ([ itm ], []);
                allItems = itm :: world.allItems
            }
        | LocationWorldEntry loc -> recurse { world with
            worldMap = mapUpdate world.worldMap loc (fun x -> x) ([], [])
            }
        | VulnerabilityWorldEntry (chr, vItems) -> (match mapLookup
world.vulnerability chr with
            | None -> recurse { world with
                vulnerability = (chr, vItems) :: world.vulnerability
                }
            | Some _ -> Left "Error: NPC's vulnerability was set twice")
```

Snippet from our parser, from `parser.mly`:

```
/* Like the worldExprs nonterminal, allows concatenation of quest actions
into a list */
questExprs:
    | quest questExprs { $1::$2 }
    | quest { [$1] }

/* Atomic quest actions
    Note that they can take both literals and variables
    both have differing implications for the AST, so we handle them all
here */
quest:
    | TknGoto TknLiteral { ActionExp (Goto, (LocationExp (LocationLiteral
$2))) }
    | TknGet TknLiteral { ActionExp (Get, (ItemExp $2)) }
    | TknKill TknLiteral { ActionExp (Kill, (CharExp (NPCLiteral $2))) }
    | TknRequire TknLiteral { ActionExp (Require, (ItemExp $2)) }
    | TknRequire TknLBrac conditionExp TknRBrac { ActionExp (Require,
CondExp $3) }
    | TknUse TknLiteral { ActionExp (Use, (ItemExp $2)) }
    | TknGoto TknVar { ActionExp (Goto, (VarExp $2)) }
    | TknGet TknVar { ActionExp (Get, (VarExp $2)) }
    | TknKill TknVar { ActionExp (Kill, (VarExp $2)) }
    | TknRequire TknVar { ActionExp (Require, (VarExp $2)) }
    | TknUse TknVar { ActionExp (Use, (VarExp $2)) }
    | TknLet TknEq builtinFunExp { LetExp ($1, $3) }
    | TknSubquestRun argumentList { RunSubquestExp ($1, $2) }
```