Description

Our solution is made up of three files, sat_server.erl, sat_worker.erl, and sockserv_serv.erl.

sat_server.erl

This file is our supervisor, whose primary functionalities are starting and managing our OTP Server using the OTP Supervisor behavior. It starts the server with start() and subsequently creates one listener using our sockserv_serv.erl server. It keeps track of how many children it has and whether a child is allowed to be spawned or not.

sockserv_serv.erl

This file is the server, which utilizes the OTP Server behavior. It's primary responsibilities are threefold: it receives and parses input via tcp, spawns and keeps track of its sat_worker(s), while at the same time communicating with the Supervisor (sat_server) on whether it should, for example, create a sibling server. Most of its functionality has come from building off of the OTP skeleton and working with various examples found on learnyousomeerlang.com. Its method of communication is via the behavior's message passing system, which also has TCP functionalities.

Each instance of sockserv_serv.erl will create another listener until the Supervisor deems it not allowed (in our case, after 8 connections). It is important to note that there will always be one extra connection that is not in use as it is meant to simply listen for new connections - this applies even to the "busy" state - where a connection will be made, it returns busy, creates another listener, and closes the connection.

Once a worker is spawned, the server listens for other connections and responds accordingly until it receives a message back from the worker and relays the response via the socket where the TCP connection originated. Workers are then killed and the server listens for more input.

sat_worker.erl

This is where the expressions is verified, evaluated, and returned via the Socket passed to it. It implements no behavior and simply receives information from its parent process.

The worker parses the input, checks if it's valid, then attempts to solve the 3-Sat expression. Our method was to find every permutation of the index values and to return the first permutation that evaluated to true and *unsat* if nothing was found. At the present, it can only accept up to 20 variables.

Requirements Fulfilled

- The server starts with start()
- The server listens for multiple connections on 3547
- The server initially responds with hello or busy (depending on the number of connections)

- The server accepts 3-Sat instances without the “dot”
- The server sends back periodic “trying” messages
- The server does not crash if a connection is terminated unexpectedly
- The server replies to unexpected input with “ignored”

Who Did What

It is tough to say exactly who did what with respect to this project because most (if not all) of it was written together. At times, one of us would be the scribe while the other would have a page open to the library looking for some hard-to-understand functionality. Other times we would split some of the easier functions up and compare results on our next meeting - where invariably argue for the efficacy of each implementation. Usually this resulted in some hybrid of both of our implementations. So again it’s tough to say. In fact, the only work that was purposefully and deliberately split up was the writing of these reports!

Testing

We tested our function against this input, and several other inputs found online (http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html - is just one of many examples). We built a testing function and also a generator.

Extra Requirements

- The server times out if no input is received for a full minute
- The server interpretes an abort message as a request to stop solving the previous instance