CS 433 (2) HW2 Report

Boris Nikulin

2019-03-06

Contents

1 B Shell Usage

The shell works similarly to other shells. The builtin commands include:

- `exit` or `Ctrl-d`
  - This command exits the shell.
- `history` or `!!`
  - This command lists the previous commands with an it's index in the history.
  - This command takes an optional integer.
    - If the integer is supplied, the history with that index is attempted to be run.
    - The integer must be positive and only reference previous commands.

2 B Shell Architecture

The entry point for the application is through the `Shell` class. `Shell` then uses the `shell::parse` namespace to parse the given line of input into a `Command`.

A `Command` is the primary unit of logic. A `Command` object determines what should happen. Since the object needs to support both actual programs for

the operating system to run as well as built in commands, which have different representations and arguments, I decided to use a sum type or tagged union to represent a `Command`. Thus, a Command either represents some built in command or a program. In Haskell syntax this looks like:

```
-- Command is a type, CmdBuiltIn is a constructor,
-- and BuiltIn is the type of the one and only
-- argument to the constructor
data Command = CmdBuiltIn BuiltIn
             | CmdProgramCmd Program
```

Built in commands also have varying forms, so I used another sum type. In Haskell syntax it would look like:

```
data BuiltIn = BuiltInNoCommand
             | BuiltInExit
             -- Maybe Int was represented in C++
             -- as negative numbers being Nothing
             -- and Just <int> as positive integers
             | BuiltInHist (Maybe Int)
             | BuiltInError String
```

I choose to represent errors as a built in command because I already had the framework to support pattern matching on built in commands already implemented in the `Shell` class.

The no command built in was really helpful for signalling parse failures as well as allowing empty input to not need special handling as an empty input becomes a first class command.

For representing real programs, the necessary data is mostly the same: the program location and arguments. Thus, I opted to use more traditional OOP constructs. `Program` is an abstract class that defines the necessary basics for all programs.

`SingleProgram` is a class that is the base class for all runnable programs. It represents a single program to be run.

Further plans include adding a `PipedProgram` as a child of `Program` that would store a list of `SinglePrograms` and then sequence them together with UNIX pipes. However, I ran out of time to get that far.

3 Notable Items

`wait()` did not work reliably consecutively. Running multiple processes in the background would cause subsequent processes to be run in the background. Waiting directly on the child through `waitpid()` fixed that issue.

Documentation is available in HTML form in the `docs` directory.

Parsing using primitive string operations is a massive pain. Haskell's monadic parser combinators are too nice and make everything else seem so lackluster.