CPSC 323 Project 2 – Parser – Documentation

Amy Parker, Amy Montalvo, Brandon Dominguez

How to run

Note: this code should work across all platforms, but has only been tested on Linux.

First, install Rust using the instructions at https://rustup.rs/

Then, choose one of two options:

Option 1: Install from crates.io

To install the parser from Crates.io, run:

```
cargo install cpsc323-parser
```

Then, to run the test cases:

```
cpsc323-parser "(id+id)*id$"
cpsc323-parser "id*id$"
cpsc323-parser "(id*)$"
```

Option 2: Run from local sources

First, download the source repository:

```
git clone https://github.com/amyipdev/cpsc323-parser
cd cpsc323-parser
```

Then use `` `cargo run` `` to run the test cases:

```
cargo run "(id+id)*id$"
cargo run "id*id$"
cargo run "(id*)$”
```

What the output means

Assuming there are no issues in lexing the input, and no special conditions with the input (such as it causing the parser to run out of tokens, run out of states, or request an illegal goto), it will print a table listing every step. The first column is the step number the parser is on. The second is the state stack, formatted as concatenated copies of “sN”, where N is the state number. The third is the current output stack, which is maintained separately from the state stack instead of being merged to provide clarity. It is formatted with spaces in between elements. The fourth is the input, which is displayed as originally provided with whatever of it is left. The fifth is the action taken at that step; acceptances yield “accept”, errors yield “error”, shifts yield “sN” where N is the state to be shifted, and “rXgY” means “reduce on rule X, goto Y”.

How this code works

The input for the parser is provided in the command line arguments, which are queried using the standard library.

A tiny lexer is then run on the input to tokenize it. While the input does not need the flexibility typically required by a parser, as we have a finite number of lexemes, they are not all the same length (id is 2, while all others are 1), requiring some basic lexing.

To catch potential non-halting conditions before they can occur, the code checks that the End token is the final token, and does not appear anywhere else. Undefined behavior (likely causing a code panic) would occur without the first of these two checks, and the second prevents accidental early-terminating inputs.

We use the Tabled library to print nice-looking tables of the stack implementation, which is done with five components: the step count (scalar), state stack (vector), output stack (vector), input (double-ended queue vector), and action (text description).

Until an error or acceptance state is reached, a loop is run. This loop matches first on the current state, then executing code within that state based on the next input. Shift actions push a number onto the state stack, pop from the token stack and push the value to the output stack, and then register the output. Reduce actions pop from the output stack, pop from the state stack, push the new non-terminal to the output stack, push the goto onto the state stack, and then register the output. Because the code for these is universal, they are handled with the custom ER1 and ER3 macros; ER1 handles production rules with 1 symbol, and ER3 handles production rules with 3 symbols (there are none with only 2 or more than 3). The S45 macro is used on states 0, 6, and 7, all of which have the same shift rules (id → s5, (→ s4). Errors set the term variable to 2, and acceptances set it to 1.

On every iteration of the loop, three conditions are checked. If there are no states or no tokens left, the parser panics; a panic is **always** a definitive case of the string not being accepted. A version of this in a real compiler would use error handling to report a specific error instead of panicking, but with no error handling apparatus, panics are a reasonable way to report parser errors. Otherwise, the term variable is checked; if it is 0, parsing continues, otherwise it terminates.

After parsing terminates, the table is then printed. If the term variable was set to 1, it reports an acceptance; otherwise, it reports a non-acceptance.