UNION COLLEGE

Assignment #2: Building the Parser

Course: *CSC-375 Compiler Design* – Professor: *Aaron Cass*
Due date: *DUE DATE*

Objectives

- Learn and implement top-down recursive descent parsing
- Complete the second part of code compilation - construct an abstract syntax tree from a stream of tokens
- Pass all tests given in *base_tests.rs*, *combination.rs*, and *edge_tests.rs*
- Work in groups of 2-3

An explanation of parsing: Converting a program into a machine-understandable language is a complicated task, and lexical analysis was just the first step. Once the lexer has done its job, we now have a list of tokens that are valid our language. However, these tokens have little semantic meaning on their own. It's the job of the parser to iterate through these tokens and construct a higher-level representation of the structure of the program.

Imagine that you're the compiler, and you come across an integer type token. What would you do with it? Such a token could indicate a variable initialization, a function declaration with an integer return type, or we could be smack in the middle of a for-loop initializer. It's impossible to know without looking at the surrounding tokens, and it's the job of the parser to use that information to find out.

Once the parsing stage is complete, the parser will have generated an abstract syntax tree representing the structure of the program. The abstract syntax tree has nodes which represent certain tokens, and children which relate to each node. For example, an if statement node might have two children: a condition node, and a body node.

The method you'll use to create this tree is called *top-down recursive descent parsing*. Imagine you're given some code with no indentation or spacing. How would you turn the stream of garbage into a readable file? You'd start at the first word and read to the right, indenting and spacing each token as you go. Recursive descent paring works a lot like this, with different indentation levels akin to the different levels of the abstract syntax tree.

It's important to note that the parser doesn't check for semantic correctness itself. If you try to assign a string to a variable of type integer, it doesn't care. However, the abstract syntax tree it creates will be invaluable for the semantic analysis that's done later on.

Parser Structure

Listing 1: mod.rs

```rust
/// Core of a parser
pub mod parser_core;
mod parse_binary_exp;
mod parse_expression;
mod parse_block;
mod parse_top_level;
mod parser_utils;
mod parse_token;
```

mod.rs:

As you saw for the lexer, the *mod.rs* defines the modules related to the parser and makes them available for other modules to see. Unlike the lexer, there are a lot of files to be worried about. Each one deals with a different level of the parser, ensuring modularity.

You don't have to do anything to this file.

parser_core.rs:

This file houses the main logic for the parser. Parsing is started by calling *let ast = Parser::parse(tokens);*, where *tokens* is the stream of tokens generated by the lexer. The *parse_router* function is the main driver of the parsing process, as it delegates to helper functions based on what token it sees. It'll be your job to write these functions.

Again, you don't have to do anything here. The functions you will write are the ones called by *parse_router*, whose signatures exist in the other files.

parse_binary_expression.rs:

Handling binary expressions is tricky, as there's quite a bit of ambiguity built in. It's recommended to convert the token stream to prefix before building the abstract syntax tree, and the function signatures are there to help you get started. This part isn't easy, so start early!

parse_expression.rs:

This part requires you to write two functions, one to parse unary expressions, and another for assignments. Make sure your implementation allows for variables to be assigned to literals, other variables, and expressions!

parse_block.rs:

This file deals with blocks of code, which are commonly found within functions and control flow statements such as if statements and loops. *parse_if_statement* and *parse_block* have been provided to you to help you get started. Use these functions as a guide.

parser_utils.rs:

There's only one function in here, and its been written for you. Put any useful functions you come up with in here!

parser_token.rs:

This deals with the parsing of individual tokens. Come up with implementations for the given function signatures.

Follow these guidelines to build the lexer structure in lexer_core.rs

1. Do not change any of the function signatures, i.e. leave all parameters, parameter types, and return types as you found them.
2. Follow all instructions given in the comments where they exist, and take any potential hints into consideration when making your design.
3. Look at the unit tests to see how the abstract syntax tree should be structured. You can pretty-print the expected abstract syntax tree in a failing test with *println!("{:#?}", expected_ast);*.
4. Use helper functions.
5. Start early. This is a lot harder than the lexer, so give yourself appropriate time to flesh out all the details.