Little Languages

Lecture 30:

Parser Generators

Open today's lecture folder, go into the lalrpop directory, and do a cold run: **$ cargo run**

We're using a library with lots of dependencies that take time to build.

Consider the following grammar

Term -> Num | "(" Term ")"

Num -> [0-9]+

- Where Num is a terminal defined as a regular expression.

- Respond to the questions on PollEverywhere as to whether each input string can be parsed given this grammar.

Parser Generators

- The language we use to *specify grammars* itself has a grammar.
  - Context-Free Grammars are made of terminals, non-terminals, production rules, and so on.

- Our Context-Free Grammars (CFG) specifies *token rules* and *syntax rules*
  - These rules give us enough information to determine if a string can be parsed or not.
  - These rules *do not* tell us what happens *when* each rule is derived from an input string.

- Syntax-Directed Translation languages augment CFGs with semantics
  - SDTs are beyond the scope of this course but we'll preview them to get a feel

- Intuition: If you write enough parsers by hand and you'll realize their structure grows predictably from the grammar. Custom "logic" is applied at specific rules to either directly interpret the rule or produce an element for the AST.

- Solution: Generate the code for a parser based on an SDT language made up of both a grammar *and* the code you want to run when a syntax rule is derived.

Early SDT tools can be traced back to...

- ... you guessed it: Bell Labs and the Unix team.

- yacc - Yet Another Compiler-Compiler
  - A LALR parser generator
  - Originally written by Stephen Johnson in the early 70s
  - Only handles syntactical analysis (parsing), programmer supplies tokenization.

- lex - Lexer (Tokenizer) Generator - 1975
  - Specify regular expressions of terminals and lex produces a tokenizer for you.
  - Originally written by Mike Lesk and Eric Schmidt
    - Yes, *that* Eric Schmidt of Google. He was a *summer intern* at Bell Labs when he made lex.

Tutorial: LALRPOP in Rust

- Today we'll walk through a brief tutorial of LALRPOP
  - It is a Rust crate capable of generating a Parser from a little SDT language

- This tutorial is inspired by (and has more detail covered) the crate's official introduction: http://lalrpop.github.io/lalrpop/README.html

- Today's repo has a baseline setup in place. Since this crate has many dependencies, it's worth going ahead and starting your first build:

$ cd 590-material-<you>/lecture/30-parser-generator/lalrpop

$ cargo build

A Simple LALRPOP File Example

Given the following Grammar:

A

Term -> Num

B

Num -> [0-9]+

In the grammar of a LALRPOP file, the code that is evaluated when a production rule is derived is on the right side of each arrow =>

Notice when the Term rule that matches a Number is derived, it evaluates to the number's value.

When the Num rule is tokenized, it is converted from a &str to an i32 and evaluates as an i32.

```
// src/v1_eval.lalrpop
use std::str::FromStr;                  // Any imports needed for semantic rules

grammar;                                // The start of the grammar
    A
pub Term: i32 = {                       // Production rule defn. pub makes usable in your code.
    <n:Num> => n,                       // Explicit "match" arm, <n:Num> says n is Num terminal
};
 B
Num: i32 = <s:r"[0-9]+"> => i32::from_str(s).unwrap(); // Terminal rule written as regex
```

Parser Generator Build Steps

- Parser Generators *emit source code* your project then relies upon just like any other source code file.

- Translating the SDT file into source code must happen before compilation.

- LALRPOP has a simple "build script" in build.rs that is invoked before your code compiles.
  - It generates Rust code in the *target* directory behind the scenes.

- Macros provided by the crate then enable you to import the parser(s).
  - The rules declared pub in the lalrpop file generate structs you can import

Using a LALRPOP Parser in Code

```
 1 #[macro_use]
 2 extern crate lalrpop_util;
 3
 4 lalrpop_mod!(pub v1_eval);
 5
 6 fn main() {
 7     let parser = v1_eval::TermParser::new();
 8     println!("{:?}", parser.parse("1"));
 9     println!("{:?}", parser.parse("(2)"));
10 }
```

- The lalrpop_mod!(pub v1_parens); macro establishes the import from its hidden location in the target directgory.
- Notice the package has a struct called **`TermParser`** that is generated *from* the lalrpop's **`pub Term`** rule.
- In this example, the first parse returns Ok(1), the second fails at unknown first token.

Extending our Grammar

```
Term   -> Num | "(" Term ")"
Num    -> [0-9]+
```

- How can we make it possible to parse pairs of parenthesis?

- Add an additional, alternate rule to Term:

```
pub Term: i32 = {
    <n:Num>                 => n,
    "(" <t:Term> ")"        => t,
};
```

Adding Direct Interpretation of Arithmetic

```
Factor -> Factor "*" Term |
          Factor "/" Term |
          Term
Term   -> Num | "(" Factor ")"
Num    -> [0-9]+
```

- Notice this grammar is not LL(1) because the Factor rule is left-recursive!
- LALR parsers operate on LR(1) grammars that are more than LL(1)
  - How? Bottom-up parsing!
  - Why did we not write a bottom up parser? Because they're *too painful* to write by hand.
  - If these kinds of things fascinate you, invest in the "Dragon Book" on Compilers by Aho, et.al.

Adding Factor to the LALRPOP Grammar

```
pub Factor: i32 = {
    <l:Factor> "*" <r:Term> => l * r,
    <l:Factor> "/" <r:Term> => l / r,
    <t:Term>                => t,
}

Term: i32 = {
    <n:Num>                 => n,
    "(" <e:Expr> ")"        => e,
};
```

```
let parser = v1_eval::FactorParser::new();
```

- There's a 1:1 correspondence with the grammar and the representation in LALRPOP.
- Notice the semantics of each of the arithmetic rules are applying the operation immediately. When we test expressions we see a single result.

Hands-on: Add Precedence to the Grammar

```
Expr   -> Expr "+" Factor |
          Expr "-" Factor |
          Factor
Factor -> Factor "*" Term |
          Factor "/" Term |
          Term
Term   -> Num | "(" Expr ")"
Num    -> [0-9]+
```

- Just like you did in thbc, precedence can be enforced via production rules in the grammar.
- Try updating your LALRPOP definitions to include addition and subtraction. Then update your main.rs file to test it out.

The generated Parser can produce an AST, too!

- In the previous examples we directly evaluated the expression as it was parsed
- Instead, the semantic rules could produce nodes of an AST like you did in thbc
- Let's take a look at how using the AST enum declarations to the right

```
#[derive(Debug)]
pub enum Expr {
    Number(i32),
    Op(Box<Expr>, Opcode, Box<Expr>),
}

#[derive(Debug)]
pub enum Opcode {
    Mul,
    Div,
    Add,
    Sub,
}
```

```
use std::str::FromStr;
use crate::ast::{Expr, Opcode};

grammar;

pub Factor: Box<Expr> = {
    <lhs:Factor> <op:FactorOp> <rhs:Term> => Box::new(Expr::Op(lhs, op, rhs)),
    <t:Term> => t,
};

FactorOp: Opcode = {
    "*" => Opcode::Mul,
    "/" => Opcode::Div,
};

Term: Box<Expr> = {
    <n: Num>              => Box::new(Expr::Number(n)),
    "(" <f:Factor> ")"    => f,
};

Num: i32 =  <s:r"[0-9]+"> => i32::from_str(s).unwrap();
```

Importing our AST enums

Notice the type produced by this rule.

Using all three components of the rule to build the AST node.

Notice here the translation from string to an enum constant variant.

Using the v2_ast Parser…

- We need to swap out the parser in our main file:

```
lalrpop_mod!(pub v2_ast);
pub mod ast;

fn main() {
    let parser = v2_ast::FactorParser::new();
    // ...
}
```

Hands-on: Add Addition/Subtraction Rules to AST

```
Expr    -> Expr ExprOp Factor |
           Factor

ExprOp -> "+" | "-"

Factor -> Factor FactorOp Term |
          Term

FactorOp-> "*" | "/"

Term    -> Num | "(" Expr ")"

Num     -> [0-9]+
```

Check-in when your parser is producing arithmetic.