CS480 Final Project Report

Team

- Rikki Gibson
- Miles Van de Wetering
- Jaden Diefenbaugh

Implemented Features

Cleaner PLY Interface

We implemented an object-oriented interface for PLY that removed all dependencies on globally-declared elements. This allows the user to interact with PLY alongside arbitrary Python code. To do so, we designed a system of classes that represent the previously necessary global elements for PLY to consume.

We decided against modifying PLY directly because the codebase for it was too messy and complex.

`Grammar` class

An encapsulation of an entire language file for PLY. Contains:

- A list of `GrammarRule` s, which contain rules for parsing and matching
- A list of `TokenPattern` s, the regex-like strings that match tokens and substrings of the input together A Grammar instance will have its FIRST and FOLLOW sets, as well as checks for if it is in LL(1) or not, performed when it is initialized.

`TokenPattern` class

This class contains all information necessary to parse a token. It includes:

- A `string` that functions as a regex to match the token
- A `function` that consumes and produces a LexToken object (the PLY class for tokens)
- A `string` name of the token

`GrammarRule` class

A `GrammarRule` represents one rule in a grammar. It has:

- A `string` representing the LHS of the rule
- A list of `Production` s that represent one production of the rule
- A `function` called when the rule is parsed

`Production` class

A `Production` is a series of symbols (each which represents a `TokenPattern` or a `GrammarRule`). It consists of:

- A list of `TokenPattern` s or/and `GrammarRule` s

To convert this class-based representation of a PLY to something the PLY engine can parse, we generate a Python `module` object from the given instances of the classes above, which we then pass to PLY. The `module` has attributes set on it based on the PLY conventions for names of lexing and parsing functions.

LL(1) Parsing (without surgery)

Our LL(1) parser has two steps: validating the grammar is indeed LL(1), and generating the LL(1) parse tree. The validation checks for three conditions: FIRST/FIRST conflicts, FIRST/FOLLOW conflicts, and left recursion. If any of the three are found, PLY's Yacc parser (LALR) is used instead. These conditions are checked in:

- `has_firstfirstconflict`
- `has_firstfollowconflict`

- `has_leftrecursion`

The parsing algorithm recursively generates the parse tree using the given grammar, a stack of the current symbols, and the list of remaining input tokens. The current list of symbols are expanded according to the LL(1) parsing algorithm, in depth-first order. Once a terminal is reached, the terminal's token (the left-most token) is consumed and returned. A non-terminal, once parsed, returns an array containing the non-terminal's name and the parse trees of each of its expanded production's symbols' parse trees. The parsing continues at the next-lowest unparsed symbols.

Who did what

While all three of us worked together on the entire project, Miles & Rikki did the majority of the PLY Interface and LL(1) Parser code. Jaden also wrote the report.

Usage

1. Install PLY.
2. `python example.py` to see our example language and syntax tree produced using our LL(1) parser.
3. See the `evil_rules` and `godly_rules` objects in `example.py` to understand how the object-oriented interface to PLY works.