CMPU 331 · Compilers · Fall 2019

Project - Parser

Due Tuesday, October 29, 2019 by 11:59pm

1 Overview

The four project milestones will direct you to design and build a compiler for the DL language, which is a simple subset of the C language. Each assignment will cover one component of the compiler: lexical analysis, parsing, semantic analysis, and code generation. Each assignment will ultimately result in a working compiler phase which can interface with other phases. You will be implementing the projects in Python.

For this assignment, you will write a parser using a parser generator called "Sly Lex-Yacc" (or "SLY"). You will describe the set of productions (parse rules) for DL in an appropriate input format, and SLY will provide the actual code for applying the productions to a stream of tokens to parse DL programs.

You may work either individually or in pairs for this project.

2 Getting Started

You will submit your code through GitHub Classroom, so the first step is to clone a copy of the skeleton files from GitHub. In Moodle, under the topic "Parsing", you'll find a link to "Parser on GitHub". Follow that link, select your team (or create a new team), and then click the "Accept this assignment" button to create your own private copy of the skeleton files. Your copy will be a GitHub repository located at `https://github.com/vassar-cs331-2019b/parser-yourusername` (substitute your GitHub username). From that page, you can copy the URL for cloning by clicking the green button "Clone or download", and then paste the URL onto the command-line as an argument to the `git clone` command.

```
$ git clone https://github.com/vassar-cs331-2019b/parser-yourusername.git
```

The clone command will create a directory named `parser-yourusername` on your machine, and in that directory you'll find the following files:

- `LICENSE` - This file contains the open source license for the code. Below the copyright line with my name on it, add another copyright line with your name. (Or, just copy this file from your lexer project repository.)
- `setup.py` - This file contains metadata about the code. Replace my name and email with yours. (Or, just copy this file from your lexer project repository.)
- `runtests.py` - This script is a simple test runner. You can call it with `python3 runtests.py` to run all the tests.
- `parser_prompt.py` - This script is an interactive prompt for the parser, which you might find useful while you're developing it. You can start the prompt by calling `python3 parser_prompt.py`. For each line of DL example code you type, it will show you the AST result of the parse, or an error for invalid streams of tokens. You can exit the prompt by typing `Control+D`.

- `sly/lex.py` and `sly/yacc.py` - These files are librares you will use in your lexer and parser. You won't make any changes to these files, they are only included in the skeleton so you don't need to install them.
- `dl/ast.py` - This library contains the class definitions for the AST node types you will use in your parser.
- `dl/lexer.py` and `tests/test_lex.py` - These are a complete implementation and tests for the lexer stage of the compiler. You can (optionally) replace them with your own implementation from the lexer project submission, together with any tests you added.
- `dl/parser.py` - This file contains a skeleton for a DL parser. There are comments indicating where you need to fill in code, but you are welcome to make any modifications to the skeleton. The skeleton is functional, and will pass two tests, but it doesn't do much.
- `tests/test_parser.py` - This file contains tests for the parser, including at least one test for each type AST node, and one test of a longer code example. The provided tests are not exaustive, you may want to add more as you work. You won't be graded on the tests you add, but testing more thoroughly can increase your confidence that the parser is working as it should. I'll incorporate the best student-added tests for the parser phase into later phases, so everyone has the benefit of them.
- `tests/simple.dl` - This file is a code example in the DL language, which is used by one of the tests.

The CS lab workstations have all the software you need installed already. If you want to work on your own laptop, you might need to install git and Python 3, if you don't have them installed already.

PyCharm (https://www.jetbrains.com/pycharm/) is one IDE for working with Python, and is already installed on the CS lab workstations (though, apparently only with a trial license). But, you can use any IDE or text editor you prefer.

3 Parser Implementation

In this assignment you will write a parser for DL. The assignment makes use of two resources: a parser generator called SLY and a collection of class definitions for abstract syntax tree (AST) nodes. The output of your parser will be an AST. You will construct this AST using semantic actions of the parser generator.

You might find it helpful to refer to these sources as you work:

- Appendix A - "The DL Language" starting on page 247 of the textbook "A Practical Approach to Compiler Construction" by Des Watson
- Table 5.1 - "Parse tree nodes" on page 120 of the textbook "A Practical Approach to Compiler Construction" by Des Watson
- SLY documentation on writing a parser: https://sly.readthedocs.io/en/latest/sly.html
- A working example of a SLY parser (you only need to pay attention to the parser part): https://github.com/dabeaz/sly/blob/master/example/calc/calc.py
- A more complete working example of a SLY parser, from our class examples: https://github.com/vassar-cs331-2019b/calc-examples/blob/master/calc/parser.py

3.1 Productions and Semantic Actions

The main work of this assignment is to implement the productions neccessary to parse DL, and also the semantic actions to build an AST as the result of the parse. This might sound like a lot, but it's actually quite simple. Implementing the productions is largely just a matter of copying the BNF grammar on page 247-248 of the textbook, and writing it in the form that SLY expects to see. Implementing the semantic actions is largely just a matter of making sure each production returns a valid AST node of some kind.

For example, the production for the `print` statement is already in the template. The BNF grammar has a line:

```
<printstatement> ::= print ( <expression> )
```

The template file for the SLY parser grammar (`dl/parser.py`) has a corresponding production definition:

```
@_('PRINT OPENPAREN expression CLOSEPAREN')
def printstatement(self, p):
    """Implement the <printstatement> production."""
    return Print(p.expression)
```

The first line of this production defintion specifies what the rule should match, using all-caps token names from the lexer like `OPENPAREN`, and lower-case rule names like `expression`, which are other productions defined in the SLY parser grammar.

The second line of the production is the Python `def` keyword, which defines functions or methods in Python. Essentially it defines a method named `printstatement` in the parser class, and says what arguments will be passed to that method: `self`, which is the current object (like `this` in Java), and `p`, which contains the parse state.

The third line is a "docstring", which simply documents the behavior of the method.

The fourth line is the semantic action, which returns an AST node for the `Print` statement. The constructor for the `Print` AST node takes a single argument, which is the AST node returned by the `expression` production. The parse state stores the results of all productions that have already successfully matched, so the way you access the result of the `expression` production is by calling `p.expression` (that is, you are calling a method named `expression` on the object named `p`).

In the end, you will have about 50 or so production definitions, which will fairly closely correspond to the list of productions in the BNF grammar, with a separate SLY production for each alternate in the BNF grammar. It's fine to just follow the BNF grammar quite literally, it will lead to a working parser and a passing grade. But, you also have the flexibility to change the grammar in any way that seems sensible. For example, since SLY is a shift-reduce parser instead of a LL(1) or recursive descent parser, you could simplify the rules in your parser by allowing left-recursion. Or, since SLY defines operator precedence, you could simplify the `<expression>`, `<term>`, `<factor>` rules into all being simple alternates of the `<expression>` rule. These sorts of changes to the grammar aren't required, and won't get you a better grade, but they can make your task of implementing the parser easier (and faster!).

If you need simple code examples to run in the interactive `parser_prompt.py`, you might find it useful to copy the small code examples in the tests. All the source code examples in the tests are valid DL programs, they're just designed to be the simplest possible programs that can successfully parse. (Some of the tests use undefined variables, which will be an error once we get to semantic analysis, but is fine at the parsing stage.)

As you are working on the production rules, you may find it helpful to do a quick debug print of the AST nodes you create. For example, you could change the production for addition, so it creates the

`BinOp` AST node, prints some debug text with the node, and then returns the node:

```
node = BinOp("PLUSOP", p.expression0, p.expression1)
print("Matched addition rule: ", node)
return node
```

When you run your parser in the tests or in `parser_prompt.py`, this change would print out something like:

```
Matched addition rule: BinOp(PLUSOP, Integer(1), Integer(2))
```

3.2 AST Nodes

The skeleton includes class definitions for a collection of AST node types to use in the parser. These are inspired by the node types defined in the textbook (on page 120), but not exactly the same. Mostly this is because Python classes are a lot more powerful and flexible than the C structs that the textbook is using to implement the AST.

The following table lists the AST classes included with the skeleton, their attributes, and what they're intended to be used for.

Class	Attributes	Use
`Integer`	`value` (the integer value of the node)	integer constants
`Variable`	`name` (the variable name)	variable identifiers
`ArrayIndex`	`name` (the array variable name), `index` (the array index)	array variables with an index
`BinOp`	`op` (the operator type), `left` (the left argument), `right` (the right argument)	Binary operators: `PLUSOP`, `MINUSOP`, `MULTIPLYOP`, `DIVIDEOP`
`RelOp`	`op` (the operator type), `left` (the left argument), `right` (the right argument)	Relational operators: `EQOP`, `NEOP`, `LTOP`, `LEOP`, `GTOP`, `GEOP`
`Assign`	`left` (the variable to assign to), `right` (the value to assign)	assignment statement
`Print`	`arg` (the argument to print)	`print` statement
`Read`	`result` (the variable to store the read value)	`read` statement
`Return`	`result` (the result to return)	`return` statement
`If`	`condition` (the condition of the `if`), `body_true` (block to run if condition is true), `body_else` (block to run if condition is false)	`if` statement
`While`	`condition` (the condition of the loop), `body` (body of the loop)	`while` statement
`Block`	`statements` (ordered list of statements)	a sequence of statements, in a $\{\ldots\}$ block
`Declarations`	`declarations` (ordered list of declarations)	a sequence of variable or function declarations, at the beginning of a program
`VariableDeclarations`	`var_type` (the type of the variables to declare, always `INT` in DL), `variables` (ordered list of variables to declare)	a sequence of variable declarations, at the beginning of a program or inside a function declaration

`FunctionDeclaration`	`name` (the function name), `body` (the body of the function), `args` (an optional list of arguments to the function), `vars` (an optional list of other variables to declare for the function)	function declaration
`FunctionCall`	`name` (the function name), `args` (an optional list of arguments to the function)	function calls
`Arguments`	`arguments` (ordered list of arguments)	a sequence of arguments for a function definiton or function call
`Program`	`body` (the body of the program), `declarations` (optional list of declarations for the program)	top-level node for the entire program

3.3 Skipped Tests

In the skeleton, many of the tests are marked as `skip`, because we don't expect these tests to pass yet. Marking the tests as skipped makes them display a clean message when you run `runtests.py`, like this:

```
... skipped 'Integers not implemented yet'
```

When you start working on the productions related to a particular test, first delete the line that marks that test to be skipped, from the test file `tests/test_parser.py`. The line will look like this:

```
@unittest.skip("Integers not implemented yet")
```

After you delete the skip marker, the test will fail, and display an ugly error message. The error message will give you some clues about what's missing from your parser. Once you've implemented all the productions needed to pass a particular test, the status report at the end of the test run will change from something like:

```
FAILED (errors=1, skipped=4)
```

To something like:

```
OK (skipped=4)
```

Passing tests is how you know you've implemented your parser correctly. When all the tests are passing with no skips, you're done.

In general, you'll find it easier to start working on the tests towards the top of the `tests/test_parser.py` file, since they test features that are easier to implement. The tests grow more complex towards the end of the file, and the very last test parses the DL code example in `tests/simple.dl`.

4 Submitting

To submit your work, `git add` any files you changed, then `git commit` them with a sensible commit message about what you changed, and finally `git push` the changes back to GitHub.

```
$ git add dl/parser.py
$ git commit -m "Implemented parsing for addition"
$ git push origin master
```

It's a good idea to commit your work frequently as you go, so you have a backup if anything goes wrong. Only the last version you submit before the deadline will be graded.