Parsing

Principles of Programming Languages

Colorado School of Mines

https://lambda.mines.edu

Activity & Overview

Learning Group Activity

Review the learning group activity with your group. Compare your solutions to the practice problems. Did anyone have any issues with the problems?

Then, as a learning group, work on a regular expression to match double-quoted string literals:

```
print("Hello, World!")
if (strcspn(cmdline, "'\"`") != strlen(cmdline)) {
printf("<text:p text:style-name=\"Glossary\">");
escape("\"1 < 2\"")
```

What you should match is shown in **bold**. Try your regular expressions at the Python REPL.

Parsing: High Level Overview

Suppose we have the following source code (which, presumably, might be for some programming language):

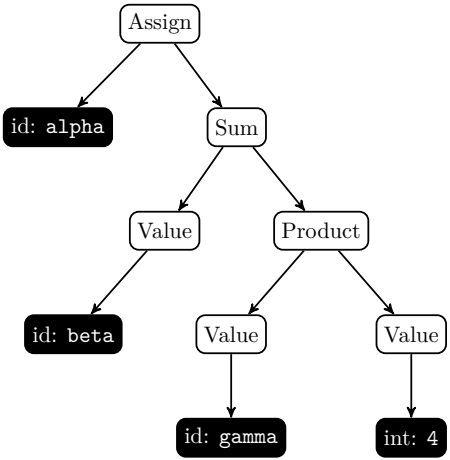
```
alpha = beta + gamma * 4
```

- How does our language implementation know what to do with this code?
- How do we determine the order of operations on this expression so that we compute `beta + (gamma * 4)` rather than `(beta + gamma) * 4`?
- How can we represent this code in memory in a way that makes it easy to evaluate or compile?
- How do we handle cases where programmers write the same expression but with different spacing or style, like this:

```
alpha=beta+gamma    *4
```

Abstract Syntax Trees

- The goal of parsing is to convert textual source code into a representation that makes it easy to interpret compile.
- We typically represent this using an **abstract syntax tree**. The abstract syntax tree for `alpha = beta + gamma * 4` is shown.
- Conveys order of operation and nesting of parentheses: Product is a child of Sum here.



Parsing: Two Stages

Parsers are typically implemented using two stages:

Lexical Analysis

During lexical analysis, the input is **tokenized** to produce a sequence of tokens from the input.

Syntactic Analysis

During syntactic analysis, the tokens from lexical analysis are formed into an abstract syntax tree.

Lexical Analysis

Lexical Analysis

During lexical analysis, we **tokenize** the input into a list tokens consisting of two fields:

- Token Type
- Data

`alpha=beta+gamma*4` $\xrightarrow{\text{LA}}$ `id:alpha, op:=, id:beta, op:+, id:gamma, op:*, int:4`

Lexical Analysis: Implementation

```
tokens_p = re.compile(r'''
    \s*(    =|\+|\*     # operators
       |    \d+         # integers
       |    \w+         # identifiers
       )\s*''', re.VERBOSE | re.DOTALL)

def tokenize(code):
    last_end = 0
    for m in tokens_p.finditer(code):
        if m.start() != last_end:
            raise SyntaxError
        tok = m.group(0)
        yield token_type(tok), tok
        last_end = m.end()
    if last_end != len(code):
```

Syntactic Analysis

Syntactic Analysis

During syntactic analysis, we turn the token stream from the lexical analysis into an abstract syntax tree.

`id:alpha, op:=, id:beta, op:+, id:gamma, op:*, int:4` $\xrightarrow{\text{SA}}$ AST

In general, there's two ways to parse a stream of tokens:

- **Top-Down:** form the node at the root of the syntax tree, then recursively form the children.
- **Bottom-Up:** start by forming the leaf nodes, then forming their parents.

Language Grammars

In order to parse a language, we need a notation to formalize the constructs of our language. We define a set of *production rules* that state what the various constructs are formed of:

$$
\begin{aligned}
\text{Assign} &\rightarrow \text{id} = \text{Sum} \\
\text{Sum} &\rightarrow \text{Sum} + \text{Product} \\
\text{Sum} &\rightarrow \text{Product} \\
\text{Product} &\rightarrow \text{Product} * \text{Value} \\
\text{Product} &\rightarrow \text{Value} \\
\text{Value} &\rightarrow \text{int} \\
\text{Value} &\rightarrow \text{id}
\end{aligned}
$$

This is actually a specific kind of context-free grammar called a LR (left-recursive) grammar. It makes it convenient for using shift-reduce parsers (coming up!)

Shift-Reduce Parsing

Shift-reduce is a type of **bottom-up** parser. We place a cursor at the beginning of the token stream, and parse each step using one of two transitions:

- **Shift:** move the cursor to the next token to the right.
- **Reduce:** match a production rule to the tokens directly to the left of the cursor, reducing them to the LHS of the production rule.

We refer to the token just to the right of the cursor as the **lookahead token**. We use the lookahead token to determine that the left of the cursor can **unambiguously** be reduced, otherwise we will shift.

Example on Whiteboard

Example shown on whiteboard of using our grammar to create an AST using shift-reduce.

Parsing SlytherLisp

Syntax of SlytherLisp

- **S-expressions** are written in parentheses and can contain any kind of expressions.
- **Numeric literals** can have an optional minus sign in front, and must have at least one digit before the optional decimal point and decimals.
- **Symbols** can consist of any amount of characters, not including `'";()` or whitespace. In addition, symbols cannot start with a digit.

Any of the preceding expressions may be **quoted** by placing a single-quote in front of them:

- Quoted S-expressions will *evaluate* as lists with each of their children quoted.
- Quoted symbols will *evaluate* to a symbol object.
- Evaluating quoted numeric literals has no special meaning (they simply behave as if they were not quoted), but the quote should still be reflected in the AST.

More Syntax

- **String literals** begin and end with a double-quote, except a escaped double-quote (\") will not end the literal.
- **Comments** begin with a semicolon and go until the end of line.
- The program may optionally begin with a shebang-line (that is, #! until the end of the first line), and this line will be ignored.

Implementing the Tokenizer

The syntax of SlytherLisp is so simple that it is not necessary to convey the type of the token when it is tokenized, so the starter code has you emit the tokens without a type.

You can change this behavior if you would like: make a new tokenize method under a different name and call it from the original tokenize method, dropping the types.

Implementing the Parser

I recommend that you start with the idea of a shift-reduce parser. You can then simplify the idea to something more specific to SlytherLisp then if you wish.

You do not have to use shift-reduce. For example, you are welcome to research and implement using a recursive descent parser (top-down) if you wish.

End of Lecture

- Class Thursday is an optional lab day.
- New learning groups next class: **you get to choose.** Email me your choices (groups of 3 to 4) by Sunday at 5 PM. If you don't send me your choice, I will assign you randomly.
- Quiz 5 will be Thursday, April 19th, and will cover regular expressions and parsing.
- Depending on how much time is left at the end of lecture, you can stick around to work on SlytherLisp if you wish.