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):

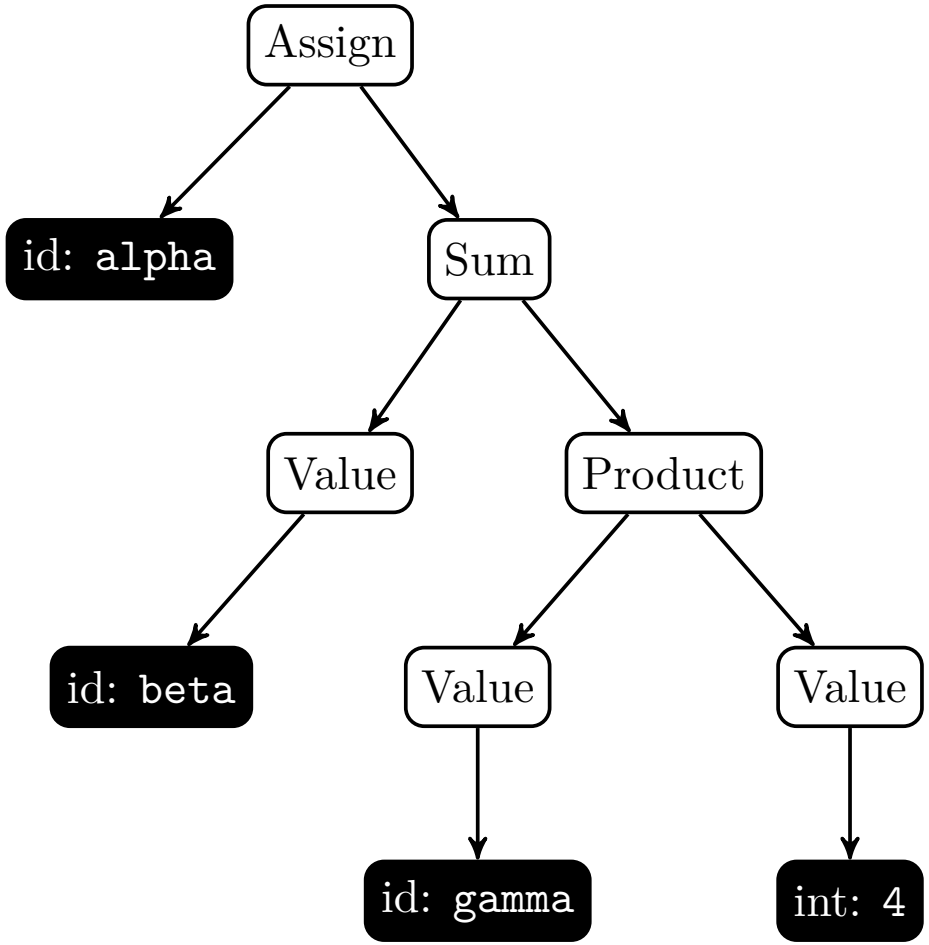
```
alpha = beta + gamma * 4
```

- 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
```

Parsing: Goal is Code to AST

- 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 (optional)

`alpha=beta+gamma*4` $\xrightarrow{\text{LA}}$ `Id(alpha), Equals, Id(beta), Plus, Id(gamma), Times, Int(4)`

- Tokens which won't appear in the AST are called **control tokens**: these control the operation of the parser.

Lexical Analysis: Implementation

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

def tokenize(code):
    for m in tokens_p.finditer(code):
        if m.group(1):
            yield Equals()
        ...
        elif m.group(5):
            yield Id(m.group(5))
        elif m.group(6):
            raise SyntaxError
```

Syntactic Analysis

Syntactic Analysis

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

`Id(alpha), Equals, Id(beta), Plus, Id(gamma), Times, 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:

$$
\begin{aligned}
\text{Assign} &\rightarrow \text{IdEqualsSum} \\
\text{Sum} &\rightarrow \text{SumPlusProduct} \\
\text{Sum} &\rightarrow \text{Product} \\
\text{Product} &\rightarrow \text{ProductTimesValue} \\
\text{Product} &\rightarrow \text{Value} \\
\text{Value} &\rightarrow \text{Int} \\
\text{Value} &\rightarrow \text{Id}
\end{aligned}
$$

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.