Programming Languages

Janyl Jumadinova

September 15, 2020
Scanning and Parsing

**Scanner**: translate source code to tokens (e.g., $< int >$, $+$, $< id >$)

- Report lexical errors like illegal characters and illegal symbols.
Scanning and Parsing

**Scanner:** translate source code to tokens (e.g., `< int >`, `+`, `< id >`)
- Report lexical errors like illegal characters and illegal symbols.

**Parser:** read token stream and reconstruct the derivation.
- Reports parsing errors – i.e., source that is not derivable from the grammar. E.g., mismatched parenthesis/braces, nonsensical statements (\(x = 1 +;\))
What is Syntax (Syntactic) Analysis?

- After lexical analysis (scanning), we have a series of tokens.
- In syntax analysis (or parsing ), we want to interpret what those tokens mean.
What is Syntax (Syntactic) Analysis?

- After lexical analysis (scanning), we have a series of tokens.
- In syntax analysis (or parsing), we want to interpret what those tokens mean.
- **Goal**: Recover the structure described by that series of tokens.
- **Goal**: Report errors if those tokens do not properly encode a structure.
Regular Expressions

- When scanning, we used regular expressions to define each token.
- Unfortunately, regular expressions are (usually) too weak to define programming languages.
Regular Expressions

- When scanning, we used regular expressions to define each token.
- Unfortunately, regular expressions are (usually) too weak to define programming languages.
- Cannot define a regular expression matching all expressions with properly balanced parentheses.
- Cannot define a regular expression matching all functions with properly nested block structure.
Regular Expressions

- When scanning, we used regular expressions to define each token.
- Unfortunately, regular expressions are (usually) too weak to define programming languages.
- Cannot define a regular expression matching all expressions with properly balanced parentheses.
- Cannot define a regular expression matching all functions with properly nested block structure.
- We need a more powerful formalism.
An alphabet is a set \( \sum \) of symbols that act as letters.

A language over \( \sum \) is a set of strings made from symbols in \( \sum \).
Formal Languages

- An **alphabet** is a set $\sum$ of symbols that act as letters.
- A **language** over $\sum$ is a set of strings made from symbols in $\sum$.
- When scanning, our alphabet is ASCII or Unicode characters. We produced tokens.
Formal Languages

- An **alphabet** is a set $\sum$ of symbols that act as letters.
- A **language** over $\sum$ is a set of strings made from symbols in $\sum$.
- When scanning, our alphabet is ASCII or Unicode characters. We produced tokens.
- When parsing, our alphabet is the set of tokens produced by the scanner.
Grammar

Grammar consists of the following::

1. a set of terminals (same as an alphabet)
2. a set of nonterminal symbols, including a starting symbol
3. a set of rules
Grammar

Grammar consists of the following:

1. a set of terminals (same as an alphabet)
2. a set of nonterminal symbols, including a starting symbol
3. a set of rules

- Strings are derived from a grammar (e.g., $S \rightarrow aS \rightarrow aaS \rightarrow aabA \rightarrow aab$
- At each step, a nonterminal is replaced by the sentential form on the right-hand side of a rule (a sentential form can contain nonterminals and/or terminals)
- Grammars generate languages
A context-free grammar (or CFG) is a formalism for defining languages.

A grammar is said to be context-free if every rule has a single nonterminal on the left-hand side.

This means you can apply the rule in any context.
One possible CFG for describing all legal arithmetic expressions using addition, subtraction, multiplication, and division

\[
E \rightarrow \text{int} \mid E \text{ Op } E \mid (E) \\
\text{Op} \rightarrow + \mid - \mid * \mid /
\]
One possible CFG for describing all legal arithmetic expressions using addition, subtraction, multiplication, and division:

\[
\begin{align*}
E & \rightarrow \text{int} \\
E & \rightarrow E \text{ Op } E \\
E & \rightarrow (E) \\
\text{Op} & \rightarrow + \\
\text{Op} & \rightarrow - \\
\text{Op} & \rightarrow * \\
\text{Op} & \rightarrow / \\
E & \rightarrow E \text{ Op } E \\
& \rightarrow E \text{ Op } (E) \\
& \rightarrow E \text{ Op } (E \text{ Op } E) \\
& \rightarrow E \times (E \text{ Op } E) \\
& \rightarrow \text{int} \times (E \text{ Op } E) \\
& \rightarrow \text{int} \times (\text{int} \text{ Op } E) \\
& \rightarrow \text{int} \times (\text{int} \text{ Op } \text{int}) \\
& \rightarrow \text{int} \times (\text{int} + \text{int})
\end{align*}
\]
Context-Free Grammar

Formally, a context-free grammar (as is the regular grammar) is a collection of four objects:

- A set of **nonterminal symbols** (or variables),
- A set of **terminal symbols**,
- A set of **production rules** saying how each nonterminal can be converted by a string of terminals and nonterminals, and
- A **start symbol** that begins the derivation.
Syntactic Analysis

\[ \text{prog} \rightarrow \{ \text{statement} \}^+ \]

\[ \text{statement} \rightarrow \text{assignment} \mid \text{loop} \mid \text{io} \]

\[ \text{assignment} \rightarrow \text{id} = \text{expression} \]

\[ \text{loop} \rightarrow \text{while} ( \text{expression} ) \text{ prog} \]

“A program is one or more statements.”

“A statement is an assignment, a loop, or an input/output command.”

“An assignment is an identifier, followed by "="; followed by an expression.”

The \{...\}+ means “one or more repetitions of the items in {...}”

In this example, “=”, “while”, “(”, and “)” are terminals
Using the BNF rules we can construct a parse tree:
Sample Parse Tree (portion)

```
<statement>
  <loop>
    while ( <expression> )
    <prog>
     <expression> <operator> <expression>
     <id> == <integer>
     a 3
     <statement>
     <assignment>
     <id> = <expression>
     found
     <integer>
     1

... while (a == 3) found = 1
```
Sample Parse Tree (failed)

Parse halts after "while" -- unable to match the "(" in the rule with a "(" in the input. An error is reported by the compiler.
Sample Parse Tree (failed)

```
{upper portion of tree omitted}
...

<statement>

<loop>

while ( <expression> ) { <prog> }

... while x <= 10
    a = x+1
```

Parse halts after "while" -- unable to match the "(" in the rule with a "(" in the input. An error is reported by the compiler.

Derivation activity: https://forms.gle/rBFCrf2sSQsoagLJ8
Grammars for Java (version 8) and Python3

- **Java**: Overview of notation used:
  https://docs.oracle.com/javase/specs/jls/se8/html/jls-2.html

- **Java**: The full syntax grammar:

- **Python**: The full grammar:
  https://docs.python.org/3/reference/grammar.html
Lex and Yacc

Programming tools for writing parsers

- **Lex** - Lexical analysis (tokenizing)
- **Yacc** - Yet Another Compiler Compiler (parsing)
SLY = Python Lex-Yacc (developed for classroom use)
https://github.com/dabeaz/sly

- Newer version of PLY: https://github.com/dabeaz/ply
  - A Python version of the lex/yacc toolset
  - Same functionality as lex/yacc, different interface
  - Consists of two Python modules: ply.lex and ply.yacc
  - Import the modules to use them
SLY

SLY = Python Lex-Yacc (developed for classroom use)
https://github.com/dabeaz/sly
- Newer version of PLY: https://github.com/dabeaz/ply
  - A Python version of the lex/yacc toolset
  - Same functionality as lex/yacc, different interface
  - Consists of two Python modules: ply.lex and ply.yacc
  - Import the modules to use them

SLY is not a code generator
SLY

- A Python version of the lex/yacc toolset
- Same functionality as lex/yacc, different interface
- Consists of two Python modules: `ply.lex` and `ply.yacc`
- Import the modules to use them