Compiler Development (CMPSC 401)

Lexical Analysis

Janyl Jumadinova

January 24, 2019
Outline

- Quick overview of basic concepts of formal grammars.
- Lexical specification of programming languages.
- Scanners and Tokens.
- Regular expressions.
Since the 1960s, the syntax of every significant programming language has been specified by a formal grammar.

Borrowed from the linguistics community - Chomsky.
Overview of Formal Languages and Automata Theory

- Starring Mr. Pig
Overview of Formal Languages and Automata Theory

- Starring Mr. Pig

- **Alphabet:** a finite set of symbols and characters
  E.g., i, k, n, o, !,
Overview of Formal Languages and Automata Theory

- Starring Mr. Pig

- **Alphabet**: a finite set of symbols and characters
  E.g., i, k, n, o, !,

- **String**: a finite, possibly empty sequence
  E.g., “oink”
Overview of Formal Languages and Automata Theory

- **Starring Mr. Pig**

- **Alphabet**: a finite set of symbols and characters
  E.g., i, k, n, o, !,

- **String**: a finite, possibly empty sequence
  E.g., “oink”

- **Language**: a set of strings (possibly empty or infinite)
  E.g., “oink!” , “oink oink!” , “oink oink oink!” , ...
Automaton - a recognizer; a machine that accepts all strings in a language (and rejects all other strings).
E.g., a pig detector: accepts all sequences of “oink”s, rejects “moo”s or “baa”s.
Automaton - a recognizer; a machine that accepts all strings in a language (and rejects all other strings).
E.g., a pig detector: accepts all sequences of “oink”s, rejects “moo”s or “baa”s.

Grammar - a generator that produced all strings in the language (and nothing else).
Language (Chomsky) hierarchy

- Regular (Type-3) languages are specified by regular expressions/grammars and finite automata (FAs) ← SCANNING
- Context-free (Type-2) languages are specified by context-free grammars and pushdown automata (PDAs) ← PARSING
- Context-sensitive (Type-1) languages
- Recursively-enumerable (Type-0) languages are specified by general grammars and Turing machines
A formal grammar for our pig language could be:

PigTalk ::= oink PigTalk (rule 1)
  | oink! (rule 2)
A formal grammar for our pig language could be:

PigTalk ::= oink PigTalk (rule 1)
| oink! (rule 2)

PigTalk can then generate, for example:

1 PigTalk ::= oink! (Rule 2)
A formal grammar for our pig language could be:

\[
PigTalk ::= oink PigTalk \text{ (rule 1)} \\
| oink! \text{ (rule 2)}
\]

PigTalk can then generate, for example:

1. \(PigTalk ::= oink! \text{ (Rule 2)}\)
2. \(PigTalk ::= oink PigTalk \text{ (Rule 1)} \)
   
   \(::= oink oink!\)
Example: Grammar for Pigese (or Pigish?)

A formal grammar for our pig language could be:

\[
\text{PigTalk ::= oink PigTalk (rule 1)} \\
\text{   | oink! (rule 2)}
\]

PigTalk can then generate, for example:

1. \[
\text{PigTalk ::= oink! (Rule 2)}
\]
2. \[
\text{PigTalk ::= oink PigTalk (Rule 1)} \\
\text{   ::= oink oink!}
\]
3. \[
\text{PigTalk ::= oink PigTalk (Rule 1)} \\
\text{   ::= oink oink PigTalk (Rule 1)} \\
\text{   ::= oink oink oink! (Rule 2)}
\]
More formally

- The rules of a grammar are called **productions**.
More formally

- The rules of a grammar are called productions.
- Rules contain:
  - **Non-terminal symbols:** grammar variables (program, statement, id, etc.)
More formally

- The rules of a grammar are called **productions**.
- Rules contain:
  - **Non-terminal symbols**: grammar variables (program, statement, id, etc.)
  - **Terminal symbols**: concrete syntax that appears in programs (a, b, c, 0, 1, if, =, (, ), ...)
More formally

- The rules of a grammar are called **productions**.
- Rules contain:
  - **Non-terminal symbols**: grammar variables (program, statement, id, etc.)
  - **Terminal symbols**: concrete syntax that appears in programs (a, b, c, 0, 1, if, =, (, ), ...)
- nonterminal ::= <sequence of terminals and nonterminals>

In a derivation, an instance of nonterminal can be replaced by the sequence of terminals and nonterminals on the right of the production.
More formally

- The rules of a grammar are called **productions**.
- Rules contain:
  - **Non-terminal symbols**: grammar variables (program, statement, id, etc.)
  - **Terminal symbols**: concrete syntax that appears in programs (a, b, c, 0, 1, if, =, (, ), ...)

`nonterminal ::= <sequence of terminals and nonterminals>`

In a derivation, an instance of nonterminal can be replaced by the sequence of terminals and nonterminals on the right of the production.

- Often there are several productions for a nonterminal derivations can choose any of them.
There are several syntax notations for productions in common use; all mean the same thing. E.g.:

\[
\text{ifStmt ::= if ( expr ) statement} \\
\text{ifStmt → if ( expr ) statement} \\
<\text{ifStmt}> ::= if ( <expr> ) <statement>
\]
A small but a more realistic example

```
program ::= statement | program statement
statement ::= assignStmt | ifStmt
assignStmt ::= id = expr ;
ifStmt ::= if (expr) statement
expr ::= id | int | expr + expr
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```
Parsing and Scanning

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

- **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 +;`)
Standard arguments about splitting functionality into independent pieces:

- Simplicity and separation of concerns
Why Separate the Scanner and the Parser?

Standard arguments about splitting functionality into independent pieces:

- Simplicity and separation of concerns
- Efficiency
But...

- Not always possible to separate cleanly.

Example: C/C++/Java type vs identifier. Things are even uglier in Fortran 77. E.g., myvar, my var, and my var are all the same identifier, keywords are not reserved, etc. Tokenizing requires context. So we hack around it somehow ... Either use simpler grammar and disambiguate later, or communicate between scanner and parser (with some semantic analysis mixed in).

Real world: Often ends up very complex and hard to follow. Compiler front-ends are sometimes referred to as "black magic."
But...

- Not always possible to separate cleanly.
  - Example: C/C++/Java type vs identifier.
  - Things are even uglier in Fortran 77.
    E.g., myvar, my var, and my var are all the same identifier, keywords are not reserved, etc. Tokenizing requires context.
But...

- Not always possible to separate cleanly.
  - Example: C/C++/Java type vs identifier.
  - Things are even uglier in Fortran 77.
    E.g., myvar, my var, and my var are all the same identifier, keywords are not reserved, etc. Tokenizing requires context.

- So we hack around it somehow ...
  - Either use simpler grammar and disambiguate later, or communicate between scanner and parser (with some semantic analysis mixed in).
  - **Real world**: Often ends up very complex and hard to follow. Compiler front-ends are sometimes referred to as “black magic”.
Typical Tokens in Programming Languages

- Operators and Punctuation
  + - * / ( ) [ ] ; : :: < <= == = != ! ...!
  Each of these is a distinct lexical class

Keywords
if while for goto return switch void ...
Each of these is also a distinct lexical class (not a string)

Identifiers (variables)
A single ID lexical class, but parameterized by actual identifier (often a pointer into a symbol table)

Integer constants
A single INT lexical class, but parameterized by numeric value

Other constants (string, floating point, boolean, ...), etc.
Typical Tokens in Programming Languages

- Operators and Punctuation
  - `+ - * / ( ) [ ] ; : :: < <= == = != ! ...!`
  - Each of these is a distinct lexical class

- Keywords
  - `if while for goto return switch void ...
  - Each of these is also a distinct lexical class (not a string)`
Typical Tokens in Programming Languages

- Operators and Punctuation
  - + - * / ( ) [ ] ; : :: < <= == = != ! ...!
  - Each of these is a distinct lexical class
- Keywords
  - if while for goto return switch void ...
  - Each of these is also a distinct lexical class (not a string)
- Identifiers (variables)
  - A single ID lexical class, but parameterized by actual identifier (often a pointer into a symbol table)
Typical Tokens in Programming Languages

- Operators and Punctuation
  - + - * / ( ) [ ] ; : :: < <= == = != ! ...!
  - Each of these is a distinct lexical class

- Keywords
  - if while for goto return switch void ...
  - Each of these is also a distinct lexical class (not a string)

- Identifiers (variables)
  - A single ID lexical class, but parameterized by actual identifier (often a pointer into a symbol table)

- Integer constants
  - A single INT lexical class, but parameterized by numeric value
Typical Tokens in Programming Languages

- Operators and Punctuation
  - `+ - * / ( ) [ ] ; : :: < <= == = != ! ...!`
  - Each of these is a distinct lexical class

- Keywords
  - `if while for goto return switch void ...`
  - Each of these is also a distinct lexical class (not a string)

- Identifiers (variables)
  - A single ID lexical class, but parameterized by actual identifier (often a pointer into a symbol table)

- Integer constants
  - A single INT lexical class, but parameterized by numeric value
  - Other constants (string, floating point, boolean, ...), etc.
Principle of Longest Match

In most languages (exception: Fortran 77), the scanner should pick the longest possible string to make up the next token if there is a choice.
In most languages (exception: Fortran 77), the scanner should pick the longest possible string to make up the next token if there is a choice.

- `return maybe != iffy;` - 5 tokens

- `RETURN ID(maybe) NEQ ID(iffy) SCOLON`
Most modern languages are free-form
- Layout doesn’t matter
- White space separates tokens

Alternatives
- Haskell, Python - indentation and layout can imply grouping
Regular Expressions used for Scanning

- Defined over some alphabet $\Sigma$.
  - For programming languages, alphabet is usually ASCII or Unicode.
- If $re$ is a regular expression, $L(re)$ is the language (set of strings) generated by $re$. 
Fundamentals of Regular Expressions (REs)

- These are the basic building blocks that other REs are built from.

<table>
<thead>
<tr>
<th>re</th>
<th>$L(re)$</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>${ a }$</td>
<td>Singleton set, for each symbol a in the alphabet $\Sigma$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>${ \varepsilon }$</td>
<td>Empty string</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>${ }$</td>
<td>Empty language</td>
</tr>
</tbody>
</table>
## Operations on REs

<table>
<thead>
<tr>
<th>re</th>
<th>$L(re)$</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>$L(r)L(s)$</td>
<td>Concatenation – r followed by s</td>
</tr>
<tr>
<td>r</td>
<td>s</td>
<td>$L(r) \cup L(s)$</td>
</tr>
<tr>
<td>r*</td>
<td>$L(r)^*$</td>
<td>0 or more occurrences of r (Kleene closure)</td>
</tr>
</tbody>
</table>
Operations on REs

<table>
<thead>
<tr>
<th>re</th>
<th>( L(re) )</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>( L(r)L(s) )</td>
<td>Concatenation – r followed by s</td>
</tr>
<tr>
<td>r</td>
<td>s</td>
<td>( L(r) \cup L(s) )</td>
</tr>
<tr>
<td>r*</td>
<td>( L(r)^* )</td>
<td>0 or more occurrences of r (Kleene closure)</td>
</tr>
</tbody>
</table>

- Precedence: \((R), R^*, R_1 R_2, R_1 | R_2\) (lowest).
- Parenthesis can be used to group REs as needed.
## Examples

<table>
<thead>
<tr>
<th>re</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>single + character</td>
</tr>
<tr>
<td>!</td>
<td>single ! character</td>
</tr>
<tr>
<td>!=</td>
<td>2 character sequence “!=“</td>
</tr>
<tr>
<td>xyzzy</td>
<td>5 character sequence “xyzzy”</td>
</tr>
<tr>
<td>(1</td>
<td>0)*</td>
</tr>
<tr>
<td>(1</td>
<td>0)(1</td>
</tr>
<tr>
<td>0</td>
<td>1(1</td>
</tr>
</tbody>
</table>
There are common abbreviations used for convenience.

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>r+</td>
<td>(rr*)</td>
<td>1 or more occurrences</td>
</tr>
<tr>
<td>r?</td>
<td>(r</td>
<td>ε )</td>
</tr>
<tr>
<td>[a-z]</td>
<td>(a</td>
<td>b</td>
</tr>
<tr>
<td>[abxyz]</td>
<td>(a</td>
<td>b</td>
</tr>
</tbody>
</table>
Abbreviations on REs

- Many systems allow abbreviations to make writing and reading definitions or specifications easier.

**Restriction**: abbreviations may not be circular (recursive) either directly or indirectly (otherwise would be not be a regular language).

- `digit ::= [0–9]` is okay
- `number ::= digit number` is not okay
Example

Possible syntax for numeric constants

digit ::= [0-9]
digits ::= digit +
number ::= digits ( . digits )?
([eE] (+ | -)? digits )?

Notice that this allows (unnecessary) leading 0s, e.g., 00045.6. (0, or 0.14 would be necessary 0s).
Example

Possible syntax for numeric constants

digit ::= [0-9]
nonzero_digit ::= [1-9]
digits ::= digit +
number ::= (0 | nonzero_digit digits?)
( . digits )?
([eE] (+ | -)? digits )?
Recognizing Regular Expressions

- Finite automata can be used to recognize languages generated by regular expressions.
- Can build by hand or automatically (tools like Lex, Flex (for compilers written in C++), and JFlex (for compilers written in Java) do this automatically, given a set of REs.)
RE Practice: https://regexone.com/

Next time:
- Building finite automata that recognize regular expressions.
- How they can be used to build scanners.