Class 14: Introduction to Grammars and Parsing

Held: Monday, 26 September 2011

Summary: Today we move from lexing to parsing. In particular, we consider the design of more complex languages through BNF grammars.

Related Pages:
- EBoard.
- Reading: Aho et al., 4.1-4.2.
- Due: Project, Phase 1: Lexer.

Notes:
- I’m back. Sorry for the extended absence, but citizenship requires it. I am likely to remain discombobulated for a few days.
- Do you want makeup classes? If so, how many?
- Are there questions on the lexical analysis assignment?
- Are there questions on the Yacc/Bison assignment?
- EC for Thursday’s Thursday extra on ???.

Overview:
- Looking Back.
- Limits of Regular Expressions.
- BNF Grammars: Form and Meaning.
- Examples.
- Looking Ahead.

Looking Back
- Our big picture goal is to write translators from high-level languages to low level languages.
- Along the way, we’ve considered what a "language" is.
  - For our purposes, it’s a set of strings.
- We’ve identified a set of steps for doing this translation.
  - You get to fill in the steps.
- We’ve done the first step of the translation, identifying the basic tokens in the input.
- We’re ready for the next step or steps.
- As in the case of regular expressions and finite automata, we will continue to look for ways to translate formal specifications to executable code.
There are some limits of regular expressions and finite automata as mechanisms for defining languages.

In particular, there are some “simple” languages that we cannot express.

Consider the language “strings of a’s and b’s with equal numbers of a’s and b’s”

This language cannot be expressed by a finite automaton.

The proof is a proof by contradiction. Suppose it could be expressed by a deterministic finite automaton, A.

- A has a finite number of states, n
- Record the sequence of states the automaton goes through given a^{n+1} as input.
- At least one state, q, must be duplicated in that sequence. Suppose it happens after a^i and a^j, with i and j different.
- If the automaton accepts a^i b^i and a^j b^j (which it should), then it will also accept a^j b^i and a^i b^j and (which it should not).

Since every regular expression and every NFA can be represented by a DFA, that language also cannot be expressed by NFAs or regular expressions.

There are a host of other similar languages that cannot be expressed by finite automata.

- a^n b^n
- palindromes
- balanced parentheses
- ...

So, what do we do? We move on to a more powerful notation.

Note that you may need pretty powerful notations.

- For example, our equal numbers of a’s and b’s may not be implementable on any machine with finite memory.

**An Introduction to Grammars**

In order to express more complicated (or more sophisticated) languages, we need a more powerful tool.

Preferably, this tool will support some form of recursion or looping.

For example, a string has equal numbers of a’s and b’s if

- the string is empty or
- the string has at least one a and at least one b and, if we remove an a and a b from the string, we end up with a string with equal numbers of a’s and b’s.

Alternatively, a string has equal numbers of a’s and b’s if

- the string is empty or
- the string is built by adding an a and a b to a string with an equal number of a’s and b’s.

The most common tool used for these recursive definitions is the Backus-Naur Form (BNF) grammar.

A grammar is a formal set of rules that specify the legal utterances in the language.
BNF grammars

- Formally, BNF grammars are four-tuples that contain
  - $\Sigma$, the alphabet from which strings are built (note that this alphabet may have been generated from the original input via a lexer). These are also called the terminal symbols of the grammar.
  - $N$, the nonterminals in the language. You can think of these as names for the structures in the language or as names for groups of strings.
  - $S$ in $N$, the start symbol. In BNF grammars, all the valid utterances are of a single type.
  - $P$, the productions that present rules for generating valid utterances.
- The set of productions is the core part of a grammar. Often, language definitions do not formally specify the other parts because they can be derived from the grammar.
- A production is, in effect, a statement that one sequence of symbols (a string of terminals and nonterminals) can be replaced by another sequence of terminals and nonterminals. You can also view a production as an equivalence: you can represent the left-hand-side by the right-hand-side.
  - If we read productions from left to right, we have a mechanism for building strings in the language.
  - If we read productions from right to left, we have a mechanism for testing language membership.
- What do productions look like? It depends on the notation one uses for nonterminals and terminals, which may depend on the available character set.
- We’ll use words in all caps to indicate terminals and words in mixed-case (or all lower case) to represent nonterminals, and stuff in quotation marks to indicate particular phrases. We’ll use ::= to separate the two parts of a rule.
- We might indicate the standard form of a Pascal program with

```
Program ::= PROGRAM
  IDENTIFIER
  OPEN_PAREN
  identifier-list
  CLOSE_PAREN
  SEMICOLON
  declaration-list
  compound-statement
  END
  PERIOD
```

- Similarly, we might indicate a typical compound statement in Pascal with

```
compound-statement ::= BEGIN
  statement-list
  END
```

- Lists of things are often defined recursively, using multiple definitions. Here’s one for non-empty statement lists. It says, “a statement list can be a statement; it can also be a statement followed by a semicolon and another statement list”.

```
```
The statements may then be defined with

\[
\text{statement} ::= \text{assignment-statement} \\
\text{statement} ::= \text{compound-statement} \\
\ldots \\
\text{assignment-statement} ::= \text{IDENTIFIER BECOMES expression} \\
\ldots
\]

At times, we may find more concise or easy to manipulate ways of writing these rules. For example, in Yacc and Bison you are expected to use each nonterminal once on the LHS but to use an or (|) to join additional options. We terminate the options with a semicolon.

\[
\text{statement-list} : = \text{statement} \\
\text{statement-list} : = \text{statement SECOLOLON statement-list} \\
\]

How do we use BNF grammars? We show derivations by starting with the start symbol and repeatedly applying productions. Eventually, we end up with a sequence of terminals.

- Any string that can be produced in this way is in the language.
- The attempt to show that a string is in a language given by a BNF grammar is called parsing that string.

Looking Ahead

- As you might guess, we’re going to do the same thing for parsing that we did for lexical analysis: We’ll consider how to go from a statement of the syntax of a language to a program that analyzes inputs to see if they match that syntax.
  - That will take a number of days.
- In this case, the translation from specification to implementation may impose some additional restrictions on the rules we can write.
- And, as in the case of `flex`, we may annotate each production with some code to execute when the production is used.