Class 05: Lexical Analysis

Held: Monday, 5 September 2011

Summary: Today we begin our consideration of lexical analysis. After exploring some general issues, we code our first lexical analyzer.

Related Pages:

- EBoard.
- Reading: Aho et al., 3.1-3.2.
- Due: Lab: Exploring Compilation.

Notes:

- Remember: The Pascal assignment is due at 11 p.m. tonight.
- Remember: The Exploring Compilation lab is due at 11 p.m. on Wednesday.
- Read for Wednesday.
- Today’s code should be in Examples/LA.
- EC for TK on Alice this Thursday.

Overview:

- The Process of Lexical Analysis.
- Detour: The Fortran Example.
- Hand-Coding a Lexical Analyzer.
- Regular Expressions.

An Introduction to Lexical Analysis

- Lexical analysis is traditionally the first “real” step in compilation. During lexical analysis, one identifies the simple tokens (also called lexemes) that make up a program.
- What are these tokens? Things like identifiers, keywords, numbers, symbols, and such.
  - Some tokens have associated semantic values, such as the name of an identifier or the value of an integer.
- During the tokenizing phase, a compiler converts a sequence of characters (sometimes thought of as a sequence of bytes or integers) to a sequence of tokens.
- During this conversion we often eliminate “unnecessary” components, such as whitespace (spaces, tabs, newlines) and comments.
  - However, there are reasons to preserve some related information, such as the line number (used in printing error messages).
- Two basic issues we need to consider in lexical analysis:
  - How do we describe the tokens of the language?
How do we implement the lexical analyzer?

- Describing tokens
  - Informally, in English-like prose
  - Formally, using BNF
  - Formally, using a simpler notation (regular expressions)

- We should have an appropriate interface for the lexical analyzer
  - How does the lexical analyzer pass tokens to the parser?
    - Typically, with a `nextToken` procedure.
  - What other information should the parser be able to request?

- There are many techniques for writing lexical analyzers (sometimes called lexers), including
  - Hand coding
  - Relying on lexical-analyzer generators
  - Relying on parser generators

- I recommend lexical-analyzer generators
  - Likely to be more efficient, since a lot of work has gone into the design of good generators.
  - Likely to be more correct, since we have automatically generated the specification of tokens to a program.
  - Likely to be more modifiable.

**Detour: The Fortran Example**

- On p. 113, Aho *et al.* have a famous example from the language Fortran.
- In its original formulation, Fortran ignored whitespace, *including in the middle of identifiers*.
- Contrast
  ```fortran
  DO 5 I = 1.25
  and
  DO 5 I = 1,25
  ```
- What is the meaning of each?
- What lessons do Aho *et al.* want you to take from this example?
- Digital legend: This typo caused the crash of Mariner I.

**Writing Lexical Analyzers by Hand**

- Even though I prefer lexical-analyzer generators, it may be useful to take a quick look at how we might hand-code a lexical analyzer.
- Let’s think about how we might write a C program to do lexical analysis for a simple language.
  - Numbers are sequences of digits.
  - Words are sequences of characters and digits.
  - Whitespace between tokens ignored (but you cannot have whitespace in the middle of a token).
  - Comments are in the standard C slash-star-slash format.
  - The operators `+,-,*,/` are permitted.
  - Everything else is invalid.
- The goal of the program is to be as efficient as possible. Hence, we don’t want to reread a character if we don’t have to.
Let’s start by designing the interface (which we’ll store in a .h file)
  • What does a token look like?
  • What does next_token look like?
  • What other operations do we provide?

Here’s a start of something in pseudocode

token readToken
  skip over whitespace and comments
  if the next character is a digit, read and return a number
  if the next character is a character, read and return a word
  if the next character is an operator, return it
  otherwise, crash and burn

We’ll think about the various parts in more detail