CSC 425 - Principles of Compiler Design I

Introduction to Lexical Analysis

Outline

- Informal sketch of lexical analysis
  - Identifies tokens in input stream
- Issues in lexical analysis
  - Lookahead
  - Ambiguities
- Specifying lexers
  - Regular Expressions

Lexical Analysis

- The goal of lexical analysis is to partition an input string into substrings where each substring is a token.
- Example:

  ```
  if (i == j)
      z = 0;
  else
      z = 1;
  ```

  is a string of characters:

  ```
  if (i == j)\n\tz = 0;else\n\tz = 1;
  ```

- A lexical analyzer is called a lexer or a scanner

Tokens

- A *token* corresponds to a set of strings
- These sets depend on the programming language
- Examples:
  - Identifiers: strings of letters or digits starting with a digit
  - Integer: a non-empty string of digits
  - Keyword (reserved word): “if”, “else”, . . .
  - Whitespace: a non-empty sequence of spaces, newlines, and tabs

What are Tokens used for?

- Classify program substrings according to role
- The output of lexical analysis is a stream of tokens
- The input to the parser is a stream of tokens
- The parser relies on token distinctions, for example, an identifier is treated differently than a keyword

Designing an Lexical Analyzer: Step 1

- Define a finite set of tokens
  - Tokens describe all items of interest
  - Choice of tokens depends on language
- Example: recall

  ```
  if (i == j)\n\tz = 0;else\n\tz = 1;
  ```

  Useful tokens:

  Integer, Keyword, Relation, Identifier, Whitespace, (,), =, ;

Designing an Lexical Analyzer: Step 2

- Describe which strings belong to each token
- Recall:
  - Identifiers: strings of letters or digits starting with a digit
  - Integer: a non-empty string of digits
  - Keyword (reserved word): “if”, “else”, . . .
  - Whitespace: a non-empty sequence of spaces, newlines, and tabs

Lexical Analyzer: Implementation

- The implementation of a lexical analyzer must do two things:
  1. Recognize substrings corresponding to tokens
  2. Return the value or *lexeme* of the token; the lexeme is the substring

Example

- Example: recall

  `if (i == j)\n\tz = 0;\nelse\tz = 1;`

- Token-lexeme groupings:
  - Identifier: i, j, z
  - Keyword: if, else
  - Relation: ==
  - Integer: 0, 1
  - Single characters: (,), =, ;

Why do Lexical Analysis?

- Simplify parsing
  - The lexer usually discards "uninteresting" tokens, for example, whitespace and comments
  - Converts data early
- Separate the logic to read source files
  - Potentially an issue on multiple platforms
  - Can optimize reading source files independently of the parser

Difficulties

- Lexical analysis can be difficult depending on the source language
- Example: in FORTRAN whitespace is insignificant
  - `VAR1` is the same as `VA R1`
  - Consider `DO 5 I = 1,25` versus `DO 5 I = 1.25`
  - Reading left-to-right, we cannot determine if `DO5I` is a variable or `DO` statement until after "," is reached
- Important points:
  - The goal is to partition the string reading left-to-right, recognizing one token at a time
  - "Lookahead" may be required to decide where the token boundaries are

Review

- The goal of lexical analysis is to:
  - Partition the input string into lexemes (the smallest program units that individually meaningful)
  - Identify the token of each lexeme
- Left-to-right scan where sometimes lookahead is required

Next

- We still need
  - A way to describe the lexemes of each token
  - A way to resolve ambiguities
    - Is `if` two variables `i` and `f` or one keyword?
    - Is == two equal signs or one operator?

Regular Languages

- There are several formalisms for specifying tokens
- Regular languages are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations

Languages

- **Definition.** Let Σ be a set of characters. A language over Σ is a set of strings of characters drawn from Σ. Σ is called the alphabet.

Examples of Languages

- Natural language
  - Alphabet: English characters
  - Language: English sentences
  - Note: not every string of English characters is an English sentence
- Programming language
  - Alphabet: ASCII
  - Language: C programs
  - Note: The ASCII character set is different from the English character set

Regular Expressions

- The lexical structure of most programming languages can be specified with regular expressions.
- *Languages* are sets of strings - we need some notation for specifying which sets we want, that is, which strings are in the set.
- A *regular expression* (RE) is a notation for a regular language
- If A is a regular expression, then we write $L(A)$ to refer to the language denoted by A.

Fundamental Regular Expressions

A	$L(A)$	Notes
a	{a}	singleton set for each symbol 'a' in the alphabet Σ
ϵ	$\{\epsilon\}$	empty string
$\varnothing$	{ }	empty language

- These are the basic building blocks of regular expressions.

Operations on Regular Expressions

A	$L(A)$	Notes
rs	$L(r)L(s)$	concatenation – r followed by s
$r\|s$	$L(r) \cup L(s)$	combination (union) – r or s
$r*$	$L(r)*$	zero or more occurrences of r (Kleene closure)

- Precedence: $*$ (highest), concatenation, $|$ (lowest)
- Parenthesis can be used to group REs as needed
- We abbreviate 'i' 'f' as 'if' (concatenation)

Examples

- $L(\text{if} \mid \text{then} \mid \text{else}) = \{\text{"if"}, \text{"then"}, \text{"else"}\}$
- $L((0 \mid 1)\ (0 \mid 1)) = \{\text{"00"}, \text{"01"}, \text{"10"}, \text{"11"}\}$
- $L(0^*) = \{\text{""}, \text{"0"}, \text{"00"}, \text{"000"}, \ldots\}$
- $L((1|0)(1|0)^*)$ = set of binary numbers with possible leading zeros

Abbreviations

Abbreviation	Meaning	Notes
$r+$	$(rr*)$	one or more occurrences
$r?$	$(r\|\epsilon)$	zero or one occurrence
$[a-z]$	$(a\|b\|\ldots\|z)$	one character in given range
$[abxyz]$	$(a\|b\|x\|y\|z)$	one of the given characters
$[\hat{}abc]$	$\overline{[abc]}$	any character except the given characters

- The basic operations generate all possible regular expressions, but common abbreviations are used for convenience.