Lexical Analysis

Lecture 5

Outline

- Informal sketch of lexical analysis
  - Identifies tokens in input string
- Issues in lexical analysis
  - Lookahead
  - Ambiguities
- Specifying lexers
  - Regular expressions
  - Examples of regular expressions

Lexical Analysis

- What do we want to do? Example:

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

- The input is just a string of characters:

```
#\tab i f ( i = = j ) #\newline  #\tab z = 0 ;
  #\newline
 e l s e #\newline #\tab  #\tab  z  =  1 ;
```

- Goal: Partition input string into substrings where the substrings are tokens or lexemes.

What's a Token?

- A syntactic category
  - In English:

    noun, verb, adjective, …

  - In a programming language:

    Identifier, Integer, Single-Float, Double-Float, operator (perhaps single or multiple character), Comment, Keyword, Whitespace, string constant, …

Defining Tokens more precisely

- Token categories correspond to sets of strings, some of them finite sets, like keywords, but some, like identifiers, unbounded.

- Identifier: (typically) *strings of letters or digits, starting with a letter. Some languages restrict length of identifier. Lisp does not require first letter…*
- Integer: *a non-empty string of digits. Bounded?*
- Keyword: **else** *or* **if** *or* **begin** *or …*
- Whitespace: *a non-empty sequence of blanks, newlines, and tabs.*

How are Tokens created?

- This is the job of the Lexer, the first pass over the source code. The Lexer classifies program substrings according to role.
- We also tack on to each token the location (line and column) of where it ends, so we can report errors in the source code by location. *(Personally, I hate this: it assumes "batch processing" is normal and makes "interactive processing" and debugging messier).*
- We read tokens until we get to an end-of-file.

How are Tokens used?

- Output of lexical analysis is a stream of tokens . . . In Lisp, we just make a list.. (token1 token2 token3 ….). [In an interactive system we would just burp out tokens as we find their ends; we don't wait for EOF]

- This stream is the input to the next pass, the parser.
  - **(parser (lexer "filename")) in Lisp.**

- Parser makes use of token distinctions
  - **An identifier FOO may be treated differently from a keyword ELSE. (Some language have no keywords. Lisp.)**

Designing a Lexical Analyzer: Step 1

- Define a finite set of tokens
  - Tokens describe all items of interest
  - Choice of tokens depends on language, design of parser
- Recall

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

- Useful tokens for this expression:

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

Designing a Lexical Analyzer: Step 2

- Describe which strings belong to each token category

- Recall:
  - Identifier: *strings of letters or digits, starting with a letter*
  - Integer: *a non-empty string of digits*
  - Keyword: *"else" or "if" or "begin" or …*
  - Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

Lexical Analyzer: Implementation

- An implementation reads characters until it finds the end of a token. (Longest possible single token). It then returns a package of up to 3 items:
  - What kind of thing is it? Identifier? Number? Keyword?
  - If it is an identifier, which one exactly? Foo? Bar?
  - Where did this token appear in the source code (line/col).
  - Whitespace and comments are skipped.

Example

- Recall:

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

- Possible Token-lexeme groupings:
  - Identifier: i, j, z
  - Keyword: if, else
  - Multi-character operator: == (not in MiniJava)
  - Integer: 0, 1
  - (,), =, ; single character operators of the same name
  - Illegal . E.g. for MiniJava, ^ % $ … everything else?

Writing a Lexical Analyzer: Comments

- The lexer usually discards "uninteresting" tokens that don't contribute to parsing, but has to keep track of line/column counts.
- If a comment can "Run off the end" it means that you must check for EOF when skipping comments too.
- Sometimes comments are embedded in comments. What to do? /* 34 /* foo */ 43 */. Java handles this "wrong".

- Sometimes compiler directives are inside comments, so comments are really NOT ignored.

/**set optimization = 0 */

True Crimes of Lexical Analysis

- Is it as easy as it sounds?
- Not quite!
- Look at some history . . .

Lexical Analysis in FORTRAN

- FORTRAN rule: Whitespace is insignificant
- E.g., `VAR1` **is the same as** `VA R1`
- Footnote: FORTRAN whitespace rule motivated by inaccuracy of punch card operators

Now we see it as a terrible design! Example

- Consider
  - `DO 5  I = 1,25`
  - `DO 5  I = 1.25`

- The first starts a loop, 25 times, running until the statement labeled "5".
  - `DO  5  I = 1  ,  25`
- The second is `DO5I = 1.25` an assignment

- Reading left-to-right, cannot tell if `DO5I` is a variable or `DO` stmt. until after "," is reached

A nicer design limits the amount of lookahead needed

Even our simple example has lookahead issues

`ii` vs. `if`

& vs. && (not in MJ…)

But this uses only one or two characters, and is certainly resolved by the time you find white space, or another character, e.g. X&Y or X&&Y. In C, consider a+++++b . Is it a++ + ++b ? Or (a++)++ +b or some other breakup?

Review

- The goal of lexical analysis is to
  - Partition the input string into lexemes
  - Identify the token type, and perhaps the value of each lexeme (probably. E.g. convert_string_to_integer("12345"))

- Left-to-right scan => lookahead sometimes required.

Next

- We need a way to describe the lexemes of each token class so that we can write a program (or build/use a program that automatically takes the description and writes the lexer!)
- Fortunately, we can start from a well-studied area in theoretical computer science, Formal Languages and Automata Theory.

Languages

Def. 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*. Pronounced SIGMA.

Why are we using Greek? So as to separate the language from the meta language. Presumably our alphabet does not include Σ as a character.)

Examples of Languages

- Alphabet = English characters
- Language = English sentences

- Not every string of English characters is an English sentence.

- Alphabet = ASCII,
- Language = C programs,

- Note: ASCII character set is different from English character set.
- UNICODE is another, much larger set, used by Java, Common Lisp.

One program's language can be another's alphabet

- An alphabet {A,B,C,D,E,—Z} can be used to form a word language: {CAT DOG EAT HAT IN THE …}.

- The "alphabet" of {CAT, DOG, EAT ….} can be used to form a different language e.g. {THE CAT IN THE HAT, ….}

Another kind of alphabet, in this case, for MJ

- Numbers
- Keywords
- Identifiers [an infinite set]
- Other tokens like + - * / ()[]{}

- Then the MJ language is defined over this pre-tokenized alphabet

Notation

- We repeat: Languages are sets of strings. Subsets of "the set of all strings over an alphabet Σ".

- Need some tools for specifying **which subset** we want to designate a particular language.
- For English, and an alphabet of words from the dictionary, we can use a grammar. (It almost works). For Java programs we can also use a grammar.

- But for now, we want to specify the set of tokens. That is, map the language of single characters into "sentences" that are tokens. Tokens have a simple structure.

Regular Languages

- There are several formalisms for specifying tokens.
- *Regular languages* are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations are possible
  - Almost powerful enough (Can't do nested comments)
  - Popular "almost automatic" tools to write programs.
- The standard notation for regular languages is *regular expressions.*

Atomic Regular Expressions

- Single character denotes a set of one string

$$'c' = \{"c"\}$$

- Epsilon character denotes a set of one 0-length string

$$\varepsilon = \{""\}$$

- Empty set is $\{\}$ = $\varnothing$ ***not the same as*** ε.
  - Size($\varnothing$)=0. Size(ε) =1

Compound Regular Expressions

- Union: If A and B are REs then…

$$A+B = \{ s \mid s \in A \text{ or } s \in B \}$$

- Concatenation of Sets ➔ Concatenation of strings

$$AB = \{ ab \mid a \in A \text{ and } b \in B \}$$

- Iteration (Kleene closure)

$$A^* = \bigcup_{i \geq 0} A^i \quad \text{where} \quad A^i = A \ldots i \text{ times} \ldots A$$

In particular, (notationally confusing concatenation with multiplication, union with addition...)

$$A^* = \varepsilon + A + AA + AAA + \ldots$$

we also sometimes use

$$A^+ = A + AA + AAA + \ldots = AA^*$$

Regular Expressions

- **Def.** The *regular expressions over* Σ are the smallest set of expressions including

$$\varepsilon$$

$$'c' \quad \text{where } c \in \Sigma$$

$$A + B \quad \text{where } A, B \text{ are rexp over } \Sigma$$

$$AB \quad " \qquad " \qquad "$$

$$A^* \quad \text{where } A \text{ is a rexp over } \Sigma$$

Syntax vs. Semantics

- This notation means this set

$$
\begin{aligned}
L(\varepsilon) &= \{""\} \\
L('c') &= \{"c"\} \\
L(A+B) &= L(A) \cup L(B) \\
L(AB) &= \{ab \mid a \in L(A) \text{ and } b \in L(B)\} \\
L(A^*) &= \bigcup_{i \geq 0} L(A^i)
\end{aligned}
$$

Example: Keyword

Keyword: *"else" or "if" or "begin" or …*

'else' + 'if' + 'begin' + ⋯

Note: 'else' denotes the same set as "e""l""s""e"

Sometimes we use convenient names for sets

Keywords= { 'else' 'if' 'then' 'when' ...}

Example: Integers

Integer: *a non-empty string of digits*

digit = '0'+'1'+'2'+'3'+'4'+'5'+'6'+'7'+'8'+'9'

integer = digit digit*

Integers are almost digit$^+$

Though, is 000 an integer?

Sometimes we use () in reg exps

For example A(B+C)

denotes the set AB+AC

Sometimes we need a language with () in it!

For example { "+" "(" ")"} denotes a set with 3 strings in it, using our metasymbols +(). Because our programming language will usually be in the same alphabet as our description of REs, we need to take some care that our metasymbols can be distinguished from our alphabet by special marks like ". Often authors leave off these marks if they think the reader will understand. If you leave such marks off in programs, you may be in trouble.

Example: Identifier

Identifier: *strings of letters or digits, starting with a letter*

- letter = 'A'+'B'+…+'Z'+'a'+…+'z'+'_'
- identifier = letter(letter + digit)*

- Is (letter$^+$ + digit)* the same?
- Is (letter* + digit*) the same?
- Is letter$^+$ (letter$^+$ + digit)* the same?

Example: Whitespace

Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

$$(' ' + '\n' + '\t')^{+}$$

Example: Phone Numbers

- Regular expressions are all around you!
- Consider (510)-642-2020.

$$
\begin{aligned}
\Sigma &= \text{digits} \cup \{\text{-},(,)\} \\
\text{exchange} &= \text{digit}^3 \\
\text{phone} &= \text{digit}^4 \\
\text{area} &= \text{digit}^3 \\
\text{phone\_number} &= \text{'('}\ \text{area}\ \text{')-'}\ \text{exchange}\ \text{'-'}\ \text{phone}
\end{aligned}
$$

Example: Email Addresses

- Consider *rootbeer@cs.berkeley.edu*

$$\Sigma = \text{letters} \cup \{.,@\}$$

$$\text{name} = \text{letter}^{+}$$

$$\text{address} = \text{name '@' name '.' name '.' name}$$

Summary

- Regular expressions describe many useful languages

- Regular languages are (exactly) those languages expressible as regular expressions.
  - We still need an implementation to take a specification of rexp R and produce a function that answers the question: Given a string *s* is

$$s \in L(R)?$$

next time.