Introduction to Lexical Analysis

Outline

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

Lexical Analysis

- What do we want to do? Example:

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

- The input is just a string of characters:

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

- Goal: Partition input string into substrings
  - where the substrings are tokens
  - and classify them according to their role

What's a Token?

- A syntactic category
  - In English:

    noun, verb, adjective, …

  - In a programming language:

    Identifier, Integer, Keyword, Whitespace, …

Tokens

- Tokens correspond to sets of strings
  - these sets depend on the programming language

- 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*

What are Tokens Used for?

- Classify program substrings according to role
- Output of lexical analysis is a stream of tokens . . .
- . . . which is input to the parser
- Parser relies on token distinctions
  - An identifier is treated differently than a keyword

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

  if (i == j)\nthen\n\tz = 0;\n\telse\n\t\tz = 1;
- Useful tokens for this expression:

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

Designing a Lexical Analyzer: Step 2

- Describe which strings belong to each token

- 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 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

- Recall:

  if (i == j)\nthen\n\tz = 0;\n\telse\n\t\tz = 1;

- Token-lexeme groupings:
  - Identifier: i, j, z
  - Keyword: if, then, else
  - Relation: ==
  - Integer: 0, 1
  - (,), =, ; single character of the same name

Why do Lexical Analysis?

- Dramatically simplify parsing
  - The lexer usually discards "uninteresting" tokens that don't contribute to parsing
    - E.g. Whitespace, Comments
  - Converts data early
- Separate out logic to read source files
  - Potentially an issue on multiple platforms
  - Can optimize reading code independently of parser

True Crimes of Lexical Analysis

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

Lexical Analysis in FORTRAN

- FORTRAN rule: Whitespace is insignificant
- E.g., **VAR1** is the same as **VA R1**

FORTRAN whitespace rule was motivated by inaccuracy of punch card operators

A terrible design! Example

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

- The first is DO 5 I = 1 , 25
- The second is DO5I = 1.25

- Reading left-to-right, the lexical analyzer cannot tell if DO5I is a variable or a DO statement until after "," is reached

Lexical Analysis in FORTRAN. Lookahead.

Two important points:

1. The goal is to partition the string
   - This is implemented by reading left-to-right, recognizing one token at a time

2. "Lookahead" may be required to decide where one token ends and the next token begins
   - Even our simple example has lookahead issues

     `i` vs. `if`

     `=` vs. `==`

Another Great Moment in Scanning History

PL/1: Keywords can be used as identifiers:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
```

can be difficult to determine how to label lexemes

More Modern True Crimes in Scanning

Nested template declarations in C++

```
vector<vector<int>> myVector

vector < vector < int >> myVector

(vector < (vector < (int >> myVector)))
```

Review

- The goal of lexical analysis is to
  - Partition the input string into *lexemes* (the smallest program units that are individually meaningful)
  - Identify the token of each lexeme

- Left-to-right scan $\Rightarrow$ lookahead sometimes 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`?
    - Is `==` two equal signs `=` `=`?

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

Def. Let Σ be a set of characters. A *language* Λ *over* Σ is a set of strings of characters drawn from Σ

(Σ is called the *alphabet* of Λ)

Examples of Languages

- Alphabet = English characters
- Language = English sentences

- Not every string on English characters is an English sentence

- Alphabet = ASCII
- Language = C programs

- Note: ASCII character set is different from English character set

Notation

- Languages are sets of strings
- Need some notation for specifying which sets of strings we want our language to contain
- The standard notation for regular languages is *regular expressions*

Atomic Regular Expressions

- Single character

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

- Epsilon

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

Compound Regular Expressions

- Union

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

- Concatenation

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

- Iteration

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

Regular Expressions

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

ε

$'c'$ where $c \in \Sigma$

$A + B$ where A, B are rexp over Σ

AB " " "

A^* where A is a rexp over Σ

Syntax vs. Semantics

- To be careful, we should distinguish syntax and semantics (meaning) of regular expressions

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

Example: Keyword

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

'else' + 'if' + 'begin' + ⋯

Note: 'else' abbreviates 'e"l"s"e'

Example: Integers

Integer: *a non-empty string of digits*

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

integer = digit digit*

Abbreviation: $A^+ = AA^*$

Example: Identifier

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

$$\text{letter} = \text{'A'} + \ldots + \text{'Z'} + \text{'a'} + \ldots + \text{'z'}$$

$$\text{identifier} = \text{letter}\ (\text{letter} + \text{digit})^*$$

Is $(\text{letter}^* + \text{digit}^*)$ the same?

Example: Whitespace

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

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

Example 1: Phone Numbers

- Regular expressions are all around you!
- Consider +30 210-772-2487

Σ = digits $\cup$ {+,–,(,)}
country = digit digit
city = digit digit
univ = digit digit digit
extension = digit digit digit digit
phone_num = '+'country' 'city'–'univ'–'extension

Example 2: Email Addresses

- Consider *kostis@cs.ntua.gr*

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

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

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

Summary

- Regular expressions describe many useful languages
- Regular languages are a language specification
  - We still need an implementation

- Next: Given a string *s* and a regular expression *R*, is

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

- A yes/no answer is not enough!
- Instead: partition the input into tokens
- We will adapt regular expressions to this goal

Implementation of Lexical Analysis

Outline

- Specifying lexical structure using regular expressions
- Finite automata
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NFAs)
- Implementation of regular expressions

  RegExp $\Rightarrow$ NFA $\Rightarrow$ DFA $\Rightarrow$ Tables

Notation

- For convenience, we will use a variation (we will allow user-defined abbreviations) in regular expression notation

- Union: $A + B \equiv A \mid B$
- Option: $A + \varepsilon \equiv A?$
- Range: $'a'+'b'+\ldots+'z' \equiv [a\text{-}z]$
- Excluded range:

  complement of [a-z] ≡ [^a-z]

Regular Expressions ⇒ Lexical Specifications

1. Select a set of tokens
   - Integer, Keyword, Identifier, LeftPar, …

2. Write a regular expression (pattern) for the lexemes of each token
   - Integer = digit +
   - Keyword = 'if' + 'else' + …
   - Identifier = letter (letter + digit)*
   - LeftPar = '('
   - …

Regular Expressions ⇒ Lexical Specifications

3. Construct R, a regular expression matching all lexemes for all tokens

$$R = \text{Keyword} + \text{Identifier} + \text{Integer} + \ldots$$
$$= R_1 + R_2 + R_3 + \ldots$$

Facts: If $s \in L(R)$ then s is a lexeme

- Furthermore $s \in L(R_i)$ for some "i"
- This "i" determines the token that is reported

Regular Expressions ⇒ Lexical Specifications

4. Let input be $x_1 \ldots x_n$
   - ($x_1 \ldots x_n$ are characters in the language alphabet)
   - For $1 \leq i \leq n$ check

     $x_1 \ldots x_i \in L(R)$?

5. It must be that

   $x_1 \ldots x_i \in L(R_j)$ for some i and j

   (if there is a choice, pick a smallest such j)

6. Report token j, remove $x1 \ldots x_i$ from input and go to step 4

How to Handle Spaces and Comments?

1. We could create a token Whitespace

   Whitespace = (' ' + '\n' + '\t')$^+$

   - We could also add comments in there
   - An input " \t\n 555 " is transformed into

     Whitespace Integer Whitespace

2. Lexical analyzer skips spaces (preferred)
   - Modify step 5 from before as follows:

     It must be that $x_k \ldots x_i \in L(R_j)$ for some j such that $x_1 \ldots x_{k-1} \in L(\text{Whitespace})$
   - Parser is not bothered with spaces

Ambiguities (1)

- There are ambiguities in the algorithm

- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$ and also $x_1 \ldots x_K \in L(R)$
- The "maximal munch" rule: Pick the longest possible substring that matches R

Ambiguities (2)

- Which token is used? What if
  - $x_1 \ldots x_i \in L(R_j)$ and also $x_1 \ldots x_i \in L(R_k)$
- Rule: use rule listed first (j if $j < k$)

- Example:
  - R_1 = Keyword and R_2 = Identifier
  - "if" matches both
  - Treats "if" as a keyword not an identifier

Error Handling

- What if

  No rule matches a prefix of input ?
- Problem: Can't just get stuck …
- Solution:
  - Write a rule matching all "bad" strings
  - Put it last
- Lexical analysis tools allow the writing of:

  $R = R_1 + \ldots + R_n$ + Error
  - Token Error matches if nothing else matches

Summary

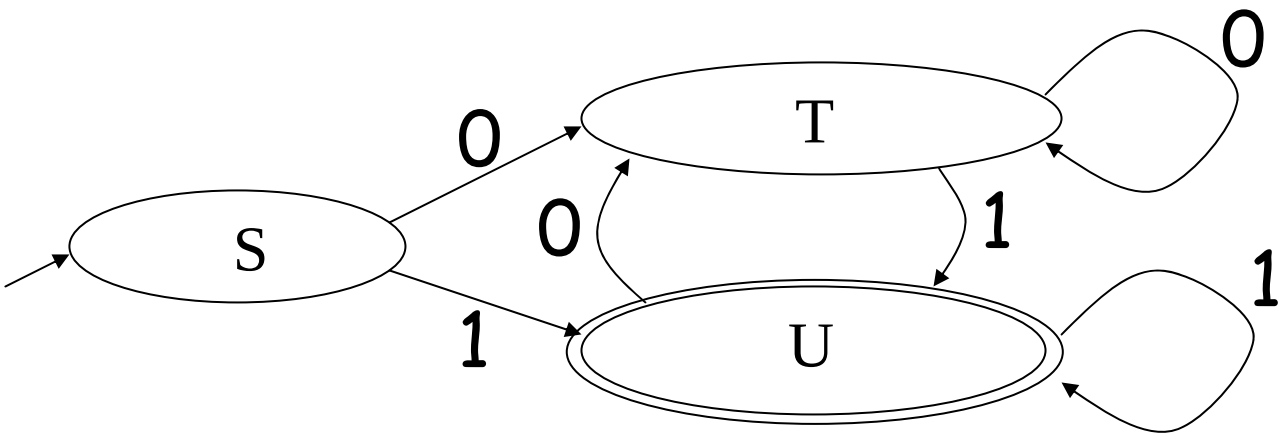
- Regular expressions provide a concise notation for string patterns
- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms known (next)
  - Require only single pass over the input
  - Few operations per character (table lookup)

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $S_i \rightarrow^a S_k$ define T[i,a] = k

- DFA "execution"
  - If in state S_i and input a, read T[i,a] = k and skip to state S_k
  - Very efficient

Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

Implementation (Cont.)

- NFA → DFA conversion is at the heart of tools such as lex, ML-Lex, flex, JLex, ...
- But, DFAs can be huge
- In practice, lex-like tools trade off speed for space in the choice of NFA to DFA conversion

Theory vs. Practice

Two differences:

- DFAs *recognize* lexemes. A lexer must return a *type of acceptance* (token type) rather than simply an accept/reject indication.

- DFAs consume the complete string and accept or reject it. A lexer must *find* the end of the lexeme in the input stream and then find the *next* one, etc.