Lexical Analysis

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:

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

- Goal: Partition input string into substrings
  - Note: humans have visual clues compiler doesn't: it just sees a sequence of bytes

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:

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

- Goal: Partition input string into substrings
  - And it's not just substrings: it's *tokens*

What's a Token?

- A syntactic category
  - In English:

    noun, verb, adjective, …

  - In a programming language:

    Identifier, Integer, Keyword, Whitespace, …

    Also: individual characters: {, }, (,), ;, …

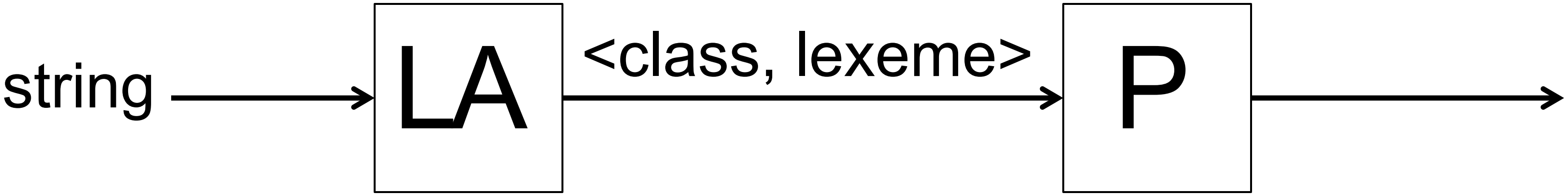
    Classifies lexeme according to its role

Tokens

- Tokens correspond to sets of strings.
  - But tokens are NOT sets of strings. Token itself is typically a tuple, e.g., <token class, lexeme>
- Identifier: *strings of letters or digits, starting with a letter*
- Integer: *a non-empty string of digits*
  - *Note possibly unusual: 001, rather than 1*
- Keyword: *"else" or "if" or "begin" or …*
- Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

What are Tokens For?

- Classify program substrings according to role
  - Which is, in effect, the goal of lexical analysis
- 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

Example

- “foo = 42” tokenized as:

  <Identifier, “foo”>

  < = , “=“>

  <Integer, “42”>

  - Note that “42” is a string. To LA, all things are strings.

Note:

- The lexical analyzer must be able to break down into tokens any string that represents a valid program in the language
- So, somehow we'll have to specify this:
  - Not only whether the string is a valid program
  - But also what each piece of the string represents (in terms of tokens)
  - AND there can be no ambiguity!

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

Example

- Recall

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

- Useful tokens for this expression:

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

- N.B., (,), =, ; are tokens, not characters, here
  - And often in token classes all by themselves

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:

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

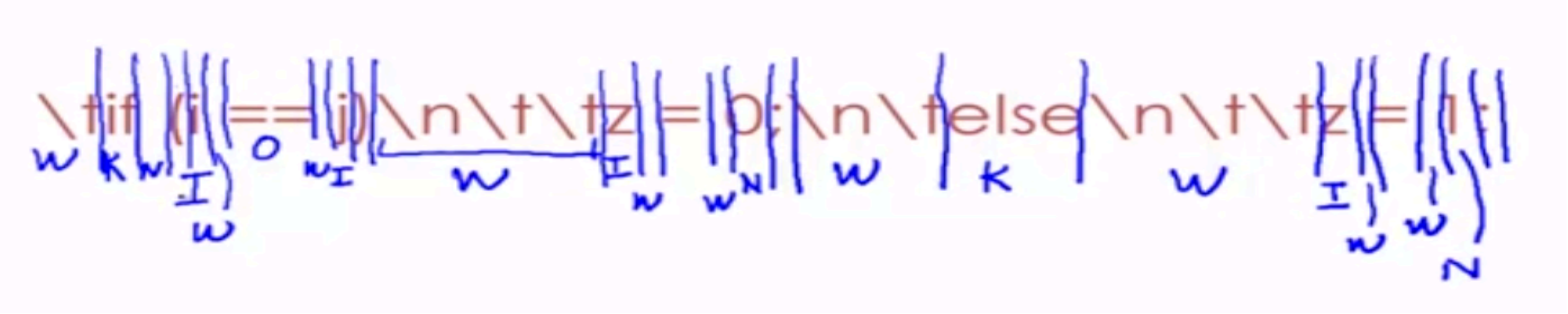
(W)hitespace
(O)perator
(K)eyword
(I)dentifier
(N)umber

Also (,),=, etc.

Example

- Recall:

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



(W)hitespace

(O)perator

(K)eyword

(I)dentifier

(N)umber

Also (,),=, etc.

Lexical Analyzer: Implementation

- The lexer usually discards “uninteresting” tokens that don’t contribute to parsing.

- Examples: Whitespace, Comments

True Crimes of Lexical Analysis

- Is it as easy as it sounds?

- Not quite!

- Look at some history . . .
  - (Note: some examples borrowed from original Dragon Book, some from new Dragon book)

Lexical Analysis in FORTRAN

- FORTRAN rule: Whitespace is insignificant
- E.g., VAR1 is the same as VA   R1
- A terrible design!
  - Their idea was that you should be able to remove all whitespace from a program and it should run exactly the same

Another Fortan Example

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

  - What is the difference here?

Another Fortan Example

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

  - What is the difference here?
  - And yet…

Another Fortan Example

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

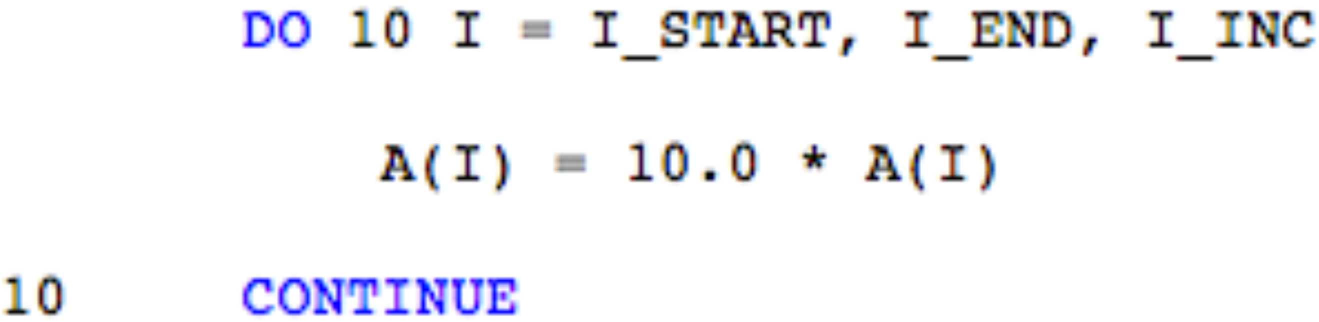
  - One of these is the header of a Fortran loop

Another Fortan Example

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

  - One of these is the header of a Fortran loop
  - The other is a variable declaration

A Fortan (77) Loop



```
       DO 10 I = I_START, I_END, I_INC
           A(I) = 10.0 * A(I)
10     CONTINUE
```

- Thanks to A.J. Miller (who was apparently a grad student at PSU around 2000)

Lexical Analysis in FORTRAN (Cont.)

- Two important points:
  1. The goal is to partition the string. This is implemented by reading left-to-write, recognizing one token at a time
  2. "Lookahead" may be required to decide where one token ends and the next token begins
     1. In this example, need to read to 11th character before knowing what token you have

Lookahead

- As you might expect, lookahead complicates the process of lexical analysis (making for a more complicated compiler)
  - So languages are designed to minimize the need for lookahead
- This being said, some lookahead is almost always required
- For example…

Lookahead

- Even our simple example has lookahead issues
  - i vs. if
  - = vs. ==

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

- Footnote: FORTRAN Whitespace rule motivated by inaccuracy of punch card operators
  - It was easy to accidentally insert blanks
  - This rule prevents having to reenter the whole card

More Lookahead

- And yet more:

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

When we read the “e” in else, we can’t know whether we have a variable name or a keyword until we’ve read through to the space after the second “e”

Lexical Analysis in PL/I

- PL/I stands for Programming Language 1
  - Was supposed to be THE programming language (at least on IBM machines)
  - Supposed to encompass every feature any programmer would ever need
  - And so was supposed to be very, very general and have very few restrictions. So…

- PL/I keywords are not reserved
  - You could have variables named same as keywords

Lexical Analysis in PL/I

- PL/I keywords are not reserved

  IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN

- So you can't know whether you have a keyword or a variable name until you've seen the entire line of code
- Which, as you might expect, makes lexical analysis in PL/I quite challenging

Lexical Analysis in PL/I (Cont.)

- PL/I Declarations:

  DECLARE (ARG1,. . ., ARGN)

- Can't tell whether DECLARE is a keyword or array reference until after the).
  - If what comes next is equal sign, it's array name
  - Requires arbitrary lookahead!

- More on PL/I's quirks later in the course . . .

Experience

- Fortran and PL/I taught folks a lot about what to do (and not do) in language design to help make lexical analysis easier.
- But…

Lexical Analysis in C++

- Unfortunately, the problems continue today

- C++ template syntax:

  `Foo<Bar>`

- C++ stream syntax:

  `cin >> var;`

- But there is a conflict with nested templates:

  `Foo<Bar<Bazz>>`

Lexical Analysis in C++

- But there is a conflict with nested templates:

  Foo<Bar<Bazz>>

- So what should the lexical analyzer do?
  - Well, for a long time C++ compilers considered it a stream operator
  - Solution: C++ eventually required a space between the two greater than signs
    - Kind of ugly to require white space to fix lexical analysis of a program

Review

- The goal of lexical analysis is to
  - Partition the input string into lexemes
  - Identify the token of each lexeme

- Left-to-right scan => 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

Formal Languages

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

(Note: not every string consisting of characters from Σ need be in the language)

Examples of Languages

- Alphabet = English characters
- Language = English sentences

- Not every string of English characters is an English sentence
  - And defining which strings of characters are valid English sentences would be tricky

- Alphabet = ASCII
- Language = C programs

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

Meaning Function

- Meaning function L maps syntax to semantics
  - Ex. Regular expression might be the syntax, semantics might be the set of strings that a regular expression represents (more later).

Notation

- Languages are sets of strings.

- Need some notation for specifying which sets we want

- The standard method for expressing regular languages is *regular expressions.*
  - But it is not the only way this can be done.

Atomic Regular Expressions

- Single character: represents the language consisting of one string

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

- Epsilon: also represents a language consisting of one string (does NOT represent the "empty language")

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

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

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

Kleene closure of A

Compound Regular Expressions

Note these are all mappings from an expression (piece of syntax) to a set of strings. The purpose of L is to clarify this.

- 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

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

Note A^0 is ε

Regular Expressions

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

ε	
$'c'$	where $c \in \sum$
$A + B$	where A, B are rexp over $\sum$
AB	" " "
A^*	where A is a rexp over $\sum$

Syntax vs. Semantics

- To be careful, we should distinguish syntax and semantics.

$$L(\varepsilon) = \{""\}$$

$$L('c') = \{"c"\}$$

$$L(A + B) = L(A) \cup L(B)$$

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

$$L(A^{*}) = \bigcup_{i \geq 0} L(A^{i})$$

Note L: Expressions -> Sets of Strings

Helps make clear what is an expression and what is a set

Note: When we write things like this

- Makes clear how we recursively apply L to decompose original compound expressions into several expressions that we compute the meaning of and then compute the sets from those separate smaller sets
- We'll come back to this. But first…

Examples

- Assume $\Sigma = \{0,1\}$
- 1^*
- $(1 + 0)1$
- $0^* + 1^*$
- $(0 + 1)^*$

Examples

- Assume $\Sigma = \{0,1\}$
- 1^*
- $(1 + 0)1$
- $0^* + 1^*$
- $(0 + 1)^*$ (a.k.a. Σ^*)

Note These are Not Unique

- Assume $\Sigma = \{0,1\}$
- 1^* same as $1^* + 1$
- $(1 + 0)1$ same as $11 + 01$
- $0^* + 1^*$
- $(0 + 1)^*$ (a.k.a. Σ^*)

Nor Are These

All denote same set of strings

0^*

$0 + 0^*$

$\varepsilon + 00^*$

$\varepsilon + 0 + 0^*$

Why Use a Meaning Function?

- Makes clear what is syntax, what is semantics
- Allows us to consider notation as a separate issue
  - Allows us to vary the syntax while keeping the semantics the same
    - Might discover that some kinds of syntax are better than others for the problems or languages which interest us
- Because expressions and meanings are not 1-1
  - As we've seen
  - Generally many more expressions than meanings

Why is separating syntax from semantics good for notation?

- Consider:

1	4	42	107
I	IV	XLII	CVII

- Turns out that Roman Numerals are really hard to use when doing things like multiplication and addition
  - Back in Roman times, very few people could do math with this
  - Algorithms were very complicated
- Arabic system eliminated this problem
  - Yet only change was the notation!

So, Notation is Important Because

- It governs how you think
- It governs the kinds of things you can say
- It governs the procedures you can use
- So: don't underestimate importance of notation!

Thus

- The importance of notation is one reason why separating syntax from semantics is beneficial
  - Ex. We can leave the notion that we’re playing with numbers out of things and just concentrate on the various ways of representing those numbers.
    - As we’ve seen, some ways of representing them might be far better than others.

Third Reason for Separating Syntax from Semantics

- For many languages in which we are interested, multiple expressions will have the same semantics
  - I.e. L is many-to-one
  - Extremely important in compilers: basis of optimization – many different programs that are functionally equivalent!

0^*

$0 + 0^*$

$\varepsilon + 00^*$

$\varepsilon + 0 + 0^*$

Note: It never works the other way

- L is never one-to-many
  - First, it would imply that L is not a function
  - More important, it would imply that one program would have more than one meaning!

Segue

- Regular expressions are simple, almost trivial
    - But they are useful!

- Reconsider informal token descriptions . . .

- And let's see how to use regular expressions to specify different aspects of programming languages

Example: Keyword

Keyword: *“else”* or *“if”* or *“begin”* or …

‘else’ + ‘if’ + ‘begin’ + . . .

Note: ‘else’ abbreviates

‘e’ ’ l’ ’ s’ ’ e’

(which is technically how you express the concatenation of these four single character regular expressions)

Example: Integers

Integer: *a non-empty string of digits*

$$\text{digit} = '0'+'1'+'2'+'3'+'4'+'5'+'6'+'7'+'8'+'9'$$

$$\text{integer} = \text{digit digit}^*$$

Why not digit*?

Abbreviation: $A^+ = AA^*$

Note: most tools allow for the naming of a regular expression (as we did with “digit” above)

Example: Integers

Integer: *a non-empty string of digits*

$$\text{digit} = \text{'0'}+\text{'1'}+\text{'2'}+\text{'3'}+\text{'4'}+\text{'5'}+\text{'6'}+\text{'7'}+\text{'8'}+\text{'9'}$$

$$\text{integer} = \text{digit digit}^*$$

Why not digit*?

Abbreviation: $A^+ = AA^*$

so integer = digit^+

Example: Identifier

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

letter = 'A' + . . . + 'Z' + 'a' + . . . + 'z'

identifier = letter (letter + digit)*

Is (letter* + digit*) the same?

Example: Identifier

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

character range, supported by most tools

letter = ‘A’ + . . . + ‘Z’ + ‘a’ + . . . + ‘z’ = [A-Z] + [a-z]

identifier = letter (letter + digit)* = [A-Za-z]

= [a-zA-Z]

Is (letter* + digit*) the same?

Example: Whitespace

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

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

Example: Whitespace

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

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

Note: we sometimes need a way of naming some characters that don’t have a very nice print representation

Typical way: some sort of escape sequences

Let’s look at some non-programming language examples

Example: Phone Numbers

- Regular expressions are all around you!
- Consider (555)-867-5309

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

Example: Email Addresses

- Consider *anyone@cs.richmond.edu*

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

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

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

Of course this assumes that email addresses only consist of letters (just to keep things simple here)

Example: Unsigned Pascal Floating Point Numbers

digit = '0' +'1'+'2'+'3'+'4'+'5'+'6'+'7'+'8'+'9'
digits = digit^{+}
opt_fraction = ('.' digits) + ε
opt_exponent = ('E' ('+' + '-' + ε) digits) + ε
num = digits opt_fraction opt_exponent

Note the use of ε to make parts of this optional

Alternative Shorthand

$$
\begin{array}{lcl}
\text{digit} & = & \text{'0'} + \text{'1'} + \text{'2'} + \text{'3'} + \text{'4'} + \text{'5'} + \text{'6'} + \text{'7'} + \text{'8'} + \text{'9'} \\
\text{digits} & = & \text{digit}^{+} \\
\text{opt\_fraction} & = & (\text{'.'}\ \text{digits}) + \varepsilon \\
\text{opt\_exponent} & = & (\text{'E'}\ (\text{'+'} + \text{'-'} + \varepsilon)\ \text{digits}) + \varepsilon \\
\text{num} & = & \text{digits opt\_fraction opt\_exponent}
\end{array}
$$

$\text{opt\_fraction} = (\text{‘.’ digits}) + \varepsilon = (\text{‘.’ digits})?$

shortcut

Alternative Shorthand

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

digits = digit^{+}

opt_fraction = ('.' digits) + ε

opt_exponent = ('E' ('+' + '-' + ε) digits) + ε

num = digits opt_fraction opt_exponent

opt_exponent = (‘E’ (‘+’ + ‘-’)? digits)?

Other Examples

- File names
- Grep tool family

Summary

- Regular expressions describe many useful languages
- Regular languages are a <u>language specification</u>
  - We still need an implementation
- Next time: Given a string *s* and a rexp *R*, is

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