40-414 Compiler Design

Lexical Analysis

Lecture 2

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
  - Where the substrings are tokens

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.

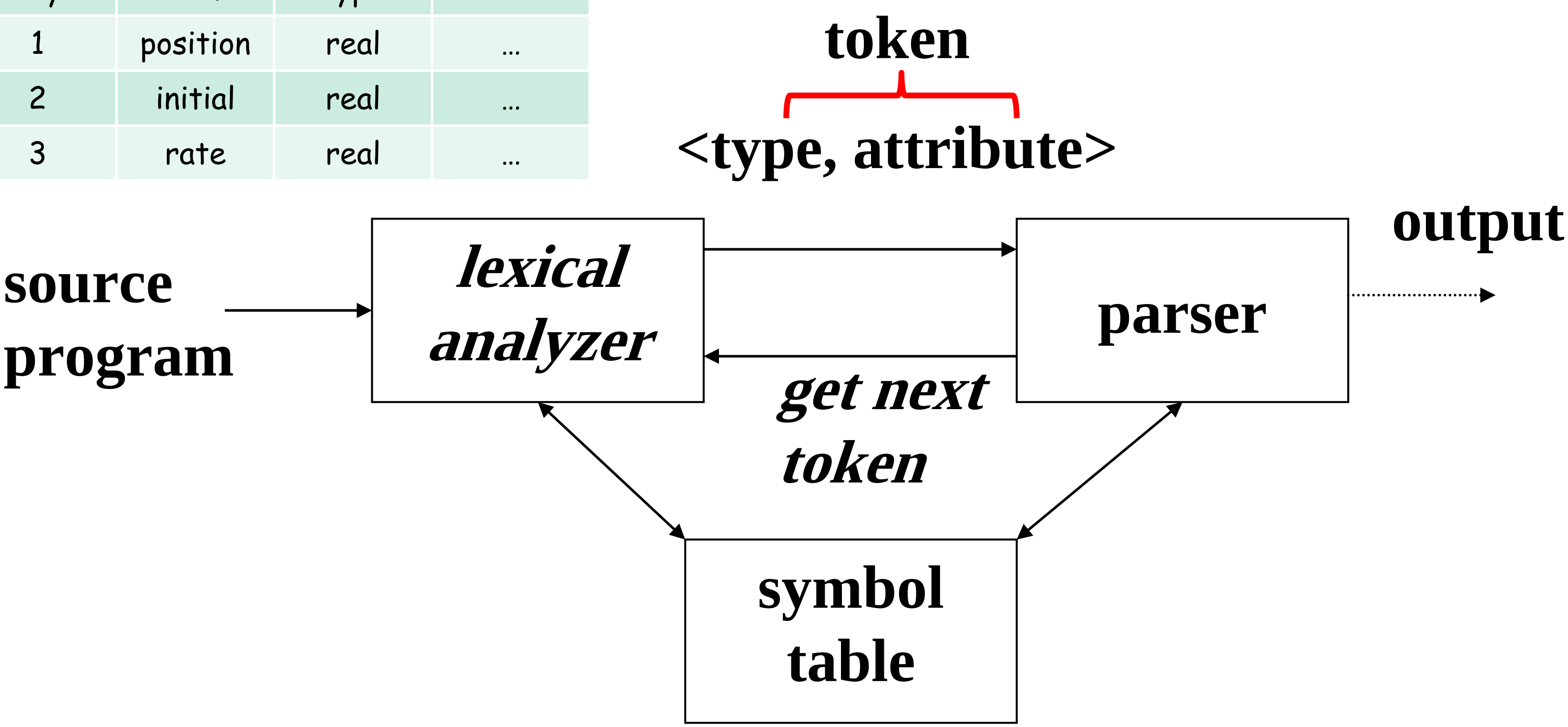
- Identifier: *strings of letters or digits, starting with a letter*: *A1*, *Foo*, *B17*
- Integer: *a non-empty string of digits*: *0*, *12*, *001*
- Keyword: *"else" or "if" or "begin" or …*
- Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

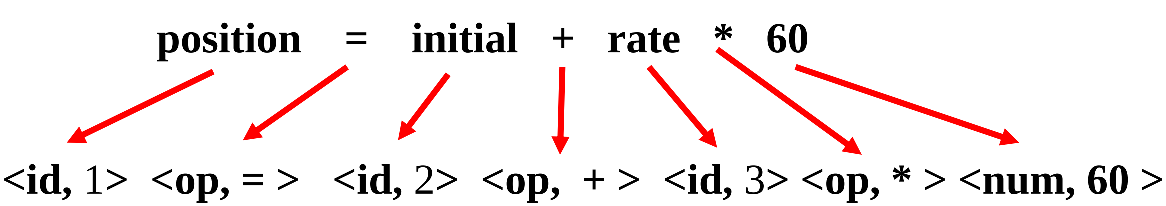
What are Tokens For?

- Classify program substrings according to role
- 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

Lexical Analyzer in Perspective

Symbol Table			
key	lexeme	type	…
1	position	real	…
2	initial	real	…
3	rate	real	…





Example

- Recall

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

  W K W (I   R I)   W   I = N ;   W   K   W   I = N ;

- Useful tokens for this expression:

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

- N.B., (,), =, ; are tokens, not characters, here

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

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

   The *lexemes*

2. Return the token class of each lexeme

   <Token class, lexeme>

   Token

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 . . .

Lexical Analysis in FORTRAN

- FORTRAN rule: Whitespace is insignificant
- E.g., VAR1 is the same as VA   R1
- A terrible design!

Example

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

Lookahead →

Fortran loop:

```
DO 5 I = 1,25
    ...
5 ...
```

Fortran assignment:

```
DO 5 I = 1.25
```

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

Lookahead

- Even our simple example has lookahead issues

  i vs. if

  = vs. ==

- Footnote: FORTRAN Whitespace rule motivated by inaccuracy of punch card operators

Lexical Analysis in PL/I

- PL/I keywords are not reserved

IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN

Keyword Keyword Keyword

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).
  - Requires arbitrary/unbounded lookahead!

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>>

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

Languages

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

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

Notation

- Languages are sets of strings.
- Need some notation for specifying which sets we want
- 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

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

Regular Expressions

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

$$\varepsilon$$

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

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

$$AB \quad \text{"} \quad \text{"} \quad \text{"}$$

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

Syntax vs. Semantics

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

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

Syntax vs. Semantics

- Why use a meaning function?
  - Makes it clear what is syntax, what is semantics.
  - Allows us to consider notation as a separate issue.
  - Because expressions and meanings are not 1-1 (ex., Roman vs Arabic numbers)

Algebraic Properties of Regular Expressions

AXIOM	DESCRIPTION
r + s = s + r	+ is commutative
r + (s + t) = (r + s) + t	+ is associative
(r s) t = r (s t)	concatenation is associative
r (s + t) = r s + r t (s + t) r = s r + t r	concatenation distributes over +
∈r = r r∈ = r	∈ Is the identity element for concatenation
r* = (r + ∈)*	relation between * and ∈
r** = r*	* is idempotent

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*

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

identifier = letter (letter + digit)*

letter = [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: Email Addresses

- Consider *anyone@cs.stanford.edu*

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

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

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

Example: Unsigned Pascal Numbers

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

digits = digit^{+}

opt_fraction = ('.' digits) $+\varepsilon$ = ('.' digits)?

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

num = digits opt_fraction opt_exponent

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 rexp *R*, is

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

Question?

For the code fragment below,
choose the correct number of tokens in each class that appear in the code fragment

```
x=0;\n\twhile (x>10){\n\tx++;\n}
```

- ○ W = 9; K = 1; I = 3; N = 2; O = 9
- ○ W = 11; K = 4; I = 0; N = 2; O = 9
- ○ W = 9; K = 4; I = 0; N = 3; O = 9
- ○ W = 11; K = 1; I = 3; N = 3; O = 9

W: Whitespace
K: Keyword
I: Identifier
N: Number
O: Other Tokens:
{ } ( ) < ++ ; =

Question?

How many distinct strings are in the language of the following regular expression:

$$(0 + 1 + \varepsilon)(0 + 1 + \varepsilon)(0 + 1 + \varepsilon)(0 + 1 + \varepsilon)$$

- ○ 31
- ○ 64
- ○ 32
- ○ 81

Question?

The language of the regular expression (abab)* is equivalent to the language of which of the following regular expressions?

Choose all that apply

- (ab)*
- (aba (baba)* b) + ε
- (ab (abab)* ab) + ε
- (a (ba)* b) + ε