Lecture 2
Lexical Analysis

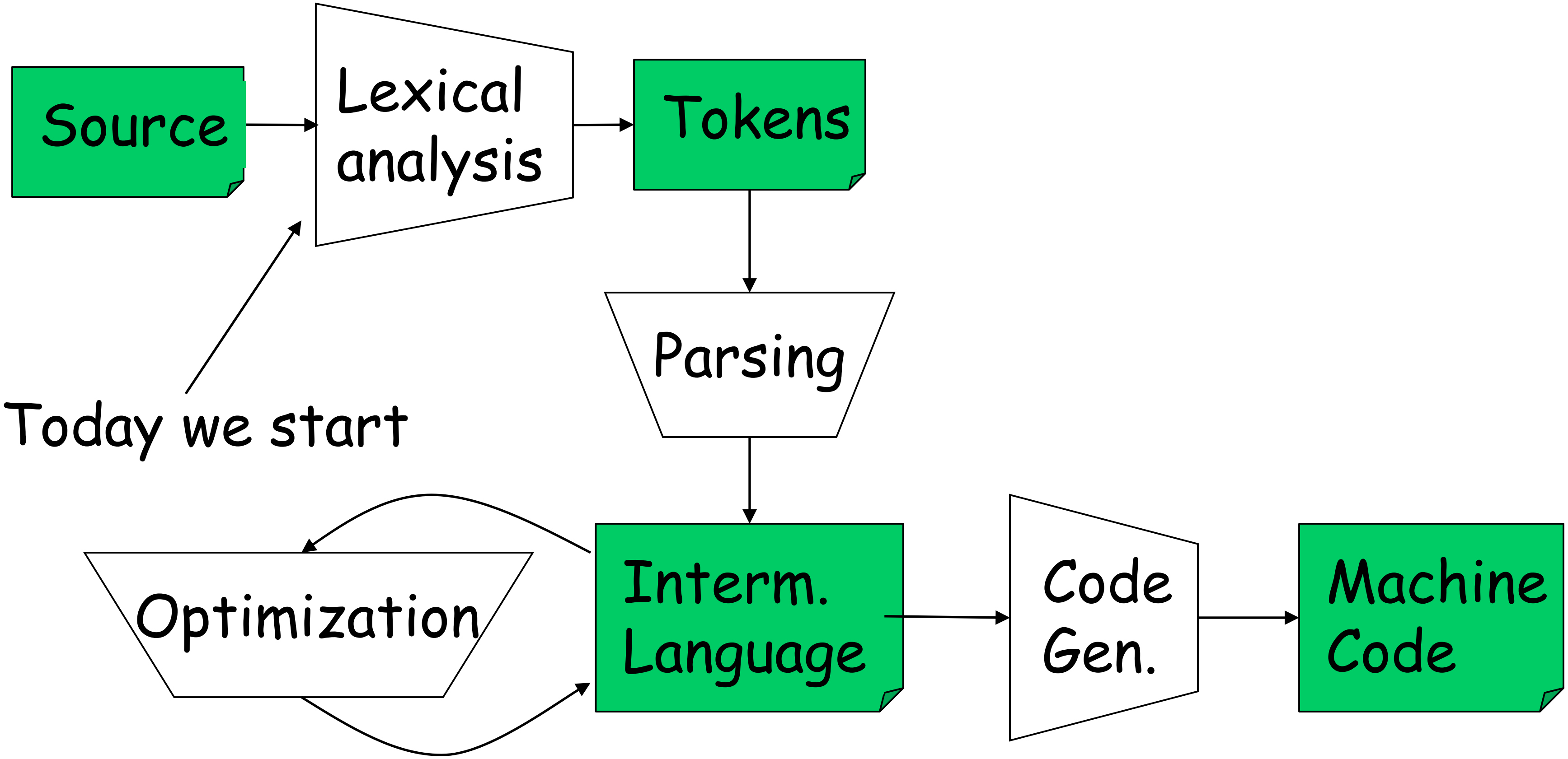
Outline

- Informal sketch of lexical analysis
  - Identifies tokens in input string

- Issues in lexical analysis
  - Lookahead
  - Ambiguities

- Specifying lexers
  - Regular expressions
  - Examples of regular expressions

The Structure of a Compiler



Lexical Analysis

- What do we want to do? Example:

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

- The input is just a sequence of characters:

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

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

What's a Token?

- Output of lexical analysis is a stream of tokens
- A token is a syntactic category
  - In English:

    noun, verb, adjective, …
  - In a programming language:

    Identifier, Integer, Keyword, Whitespace, …

Tokens

- Tokens correspond to <u>sets of strings</u>:
  - Identifiers: *strings of letters or digits, starting with a letter*
  - Integers: *non-empty strings of digits*
  - Keywords: *"else" or "if" or "begin" or …*
  - Whitespace: *non-empty sequences 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 the 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

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

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;

- Token-lexeme pairs returned by the lexer:
  - (Whitespace, “\t”)
  - (Keyword, “if”)
  - (OpenPar, “(“)
  - (Identifier, “i”)
  - (Relation, “==“)
  - (Identifier, “j”)
  - …

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

Lexical Analysis in FORTRAN (Cont.)

- 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

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

- PL/I Declarations:

  DECLARE (ARG1,. . ., ARGN)

- Can't tell whether DECLARE is a keyword or array reference until after the).
  - Requires arbitrary lookahead!
- More on PL/I's quirks later in the course . . .

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

(Σ is called the *alphabet*)

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 we want
- The standard notation for regular languages is regular expressions.

Atomic Regular Expressions

- Single character: 'c'

  'c' = { "c" } (for any $c \in \Sigma$)

- Epsilon: ε

  ε = { "" }

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^*

  $A^* = \mathrm{U}_{i>=0} A^i$ where $A^i = A \ldots i \text{ times} \ldots A$

Examples

- L('i' 'f') = { "if" } (we will abbreviate 'i' 'f' as 'if')
- 'if' | 'then' | 'else' = { "if", "then", "else"}
- '0' | '1' | … | '9' = { "0", "1", …, "9" }

  (note the … are just an abbreviation)
- ('0' | '1') ('0' | '1') = { "00", "01", "10", "11" }
- '0'* = { "", "0", "00", "000", …}
- '1' '0'* = { strings starting with 1 and followed by 0's }

  (note L means Language, sometimes we omit it)

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 where A, B are rexp over Σ

  A* where A is a rexp over Σ

Regular Expressions and Regular Languages

- Each regular expression is a notation for a regular language (a set of words)
- If A is a regular expression then we write $L(A)$ to refer to the language denoted by A

Syntax vs. Semantics

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

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

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

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

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

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

Segue

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

- Reconsider informal token descriptions . . .

Example: Keyword

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

'else' + *'if'* + *'begin'* + …

('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'
number = digit digit*

Abbreviation: $A^+ = A\ A^*$

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 as (letter + digit) * ?

Example: Whitespace

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

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

Example: Phone Numbers

- Regular expressions are all around you!
- Consider (510) 643-1481

Σ = { 0, 1, 2, 3, …, 9, (,), - }

area = digit^3

exchange = digit^3

phone = digit^4

number = '(' area ')' exchange '-' phone

Example: Email Addresses

- Consider anyone@cs.stanford.edu

Σ = letters $\cup$ { ., @ }

name = letter$^+$

address = name '@' name '.' name '.' 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 rexp R, is

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

Written Assignment

- Write regular expressions for the following languages over the alphabet $\Sigma = \{0, 1\}$:
- (a) The set of all strings which start and end with the same digit.
- (b) The set of all strings representing a binary number where the sum of its digits is even.
- (c) The set of all strings that contain the substring 10100.