Lexical Analysis

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

int a = 5; → → [int] [a] [=] [5] [;]

Tokens are things like:

- Keywords: if, while, return
- Identifiers: x, myVariable
- Operators: +, ==
- Literals: 42, "hello"

LookAhead

Lexer needs to **peek ahead** at more characters to decide the correct token.

Ex: In <=, the lexer needs to see both < and = to recognize it's one token, not two.

when a sequence of characters can match **more than one token pattern**.

Ex: ififelse → if, ifelse or if, if, else?

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:

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

- Goal: Partition input string into substrings
  - Where the substrings are tokens

What's a Token?

- A syntactical 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
  - 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

Substring	Role / Token Type	Example
int	Keyword	if, return
a, x1	identifier	variable Names

- Output of lexical analysis is a stream of tokens . . .

int a = 5; → <KEYWORD, "int"> <IDENTIFIER, "a"> <ASSIGN, "="> <INT_LITERAL, 5> <SEMICOLON, ";">

- . . . which is input to the parser

The **parser** receives this stream of tokens as input.

It **uses grammar rules** (CFG) to understand **structure** of program.

- Parser relies on token differences
  - An identifier is treated differently than a keyword
    - **if** is **not** an identifier. It starts a **conditional statement**.
    - **sum** is an **identifier** and could represent a **variable**.

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)\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:

    \tif (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
- Footnote: 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 — **In 1ST,** compiler must recognize DO, 5, I, =, 1, and 25 as **separate tokens**.
- The second is DO5I = 1.25 — **In 2nd**, without space after DO, it becomes DO5I, which is a valid **variable name**.

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

Poor language design can **complicate lexical analysis**.

Ambiguities should be avoided by **clear syntax rules**,

Lexical Analysis in FORTRAN. Lookahead

- 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
     - Even our simple example has lookahead issues

       i vs. if

       = vs. ==

Another Great Moment in Scanning

- 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

Why This is Difficult:

- **Keywords (IF, THEN, ELSE) are not reserved** in PL/1 — they **can be used as variable names**.
- This makes it **extremely hard** for the **lexer** and **parser** to distinguish between:
  - a **keyword** (control structure)
  - an **identifier** (variable)

More Modern True Crimes in Scanning

- Nested template declarations in C++

vector<vector<int>> myVector

vector < vector < int > > myVector

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

spaces were added between the angle brackets: **> >**

syntactically incorrect → Misuses parentheses () and incorrectly treats >> inside them

Review

The goal of lexical analysis is to

– Partition input string into lexemes (smallest program units that are individually meaningful)

– Identify the token of each lexeme

- Left-to-right scan ⇒ lookahead sometimes required

Note

- 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

1. Simple and Useful Theory

- Regular languages are defined by **regular expressions** and **finite automata**.
- The mathematical foundation is well-established, enabling formal reasoning and correctness proofs.
- They can describe common token patterns like identifiers, numbers, keywords, and operators.

2. Easy to Understand

- Regular expressions like [a-z, A-Z_][a-z, A-Z, 0-9_]* (for identifiers) are intuitive.
- Syntax rules for tokens can be written and read easily even by students or developers without deep formal training.

3. Efficient Implementations

- Lexical analyzers based on finite state machines can scan and classify input in **linear time (O(n))**.
- Tools like **Lex/Flex** automatically generate efficient C/C++ lexers from regular expressions.

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*)

- 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

- Just combining letters doesn't make valid English.

  Ex: "xzg klm." is **not** an English sentence.

- Not every string of ASCII characters is a valid C program!

Ex: int 3$abc is not a syntactically valid C program.

Notation or Representation

- 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: 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

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

Regular Expressions: **Def.** The regular expressions over Σ are the smallest set of expressions including

$$\varepsilon$$

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

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

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

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

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' + L

Note: else abbrevtes 'e’ 'l’ 's’ 'e'

Example: Integers and Identifier

- Integer: a non-empty string of digits

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

  integer = digit digit *

  Abbreviation: *A+ = AA**

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

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

Example 1: Phone Numbers

- Regular expressions are all around you!
- Consider +46(0)18-471-1056

$$
\begin{aligned}
\Sigma &= \text{digits} \cup \{+,-,(,)\} \\
\text{country} &= \text{digit}\ \ \text{digit} \\
\text{city} &= \text{digit}\ \ \text{digit} \\
\text{univ} &= \text{digit}\ \ \text{digit}\ \ \text{digit} \\
\text{extension} &= \text{digit}\ \ \text{digit}\ \ \text{digit}\ \ \text{digit} \\
\text{phone\_num} &= \text{`+'country'('0')'city'-'univ'-'extension}
\end{aligned}
$$

Consider kostis@it.uu.se
→→→→→→

$$
\begin{aligned}
\Sigma &= \text{letters} \cup \{.,@\} \\
\text{name} &= \text{letter}^{+} \\
\text{address} &= \text{name '@' name '.' name '.' name}
\end{aligned}
$$

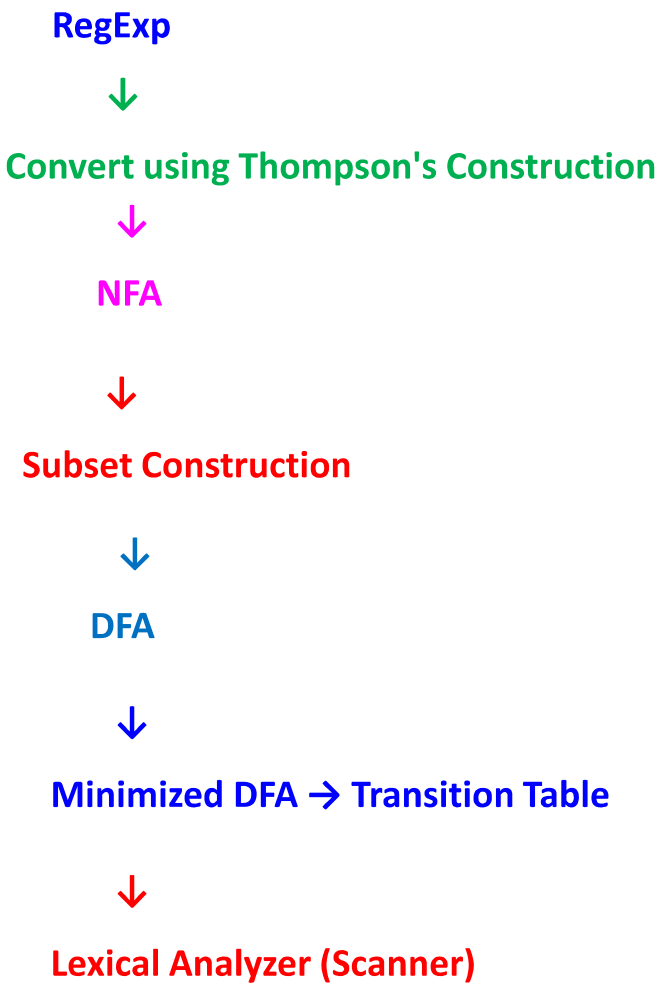
Compilers

Implementation of Lexical Analysis
Part 2

Outline

- Specifying lexical structure using regular expressions
- Finite automata
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NFAs)
- Implementation of regular expressions
- RegExp ⇒ NFA ⇒ DFA ⇒ Tables

Transition Tables (used by lexical analyser) : **(1)** Read source code character by character, **(2)** Recognize tokens using state transitions



Notation

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

- Union: A + B ≡ A | B
- Option: A + ε ≡ A?
- Range: 'a'+'b'+...+'z' ≡ [a-z]
- Excluded range:
  complement of [a-z] ≡ [^a-z]

Regular Expressions in Lexical Specification

Previous class: a specification for the predicate s ∈ L(R)

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

Breaking Down: s ∈ L(R)

- The notation **s ∈ L(R)** means: **The string s is in the language defined by the regular expression R.**

Breaking it down:

- **s** = a string (like "abc", "1010", etc.)
- **R** = a **regular expression** (like a(b|c)*, [a-z]+, etc.)
- **L(R)** = the **language** represented by the regular expression R, i.e., the **set of all strings** that match the pattern described by R.

- If **R = (a|b)abb,* Then **L(R)** = all strings of a and b ending with abb, like: (1) "abb“, (2) "aabb“, (3) "baabb"
- So if **s = "aabb"**, then **s ∈ L(R)** → satisfied the rule (because "aabb" matches the pattern)
- But
  s = "aba", then **s ∉ L(R)** → not satisfied the rule (because it doesn't end with "abb")

Regular Expressions ⇒ Lexical Spec. (1)

1. Select a set of tokens
   - Integer, Keyword, Identifier, OpenPar, ...
2. Write a regular expression (pattern) for then lexemes of each token
   - Integer = digit +
   - Keyword = 'if' + 'else' + ...
   - Identifier = letter (letter + digit)*
   - OpenPar = '('
   - ...

Regular Expressions ⇒ Lexical Spec. (2)

3. Construct R, 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 Spec. (3)

4. Let input be $x_1 \ldots x_n$
   - ($x_1 \ldots x_n$ are characters)
   - For $1 \le i \le n$ 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 j

   (if there is a choice, pick a smallest such j)
6. Remove $x_1 \ldots x_i$ from input and go to previous step

(4) Does the substring from the start to position **i** match any regular expression R that defines a valid token?

(5) Here, multiple regular expressions $R_1, R_2, \ldots, R_k$ define different types of tokens (*ex: identifiers, numbers, keywords*).

(6)

- Once a valid token is recognized ($x_1 \ldots x_i \in L(R_j)$), extract it.
- Remove it from the input.
- Restart the process for the **remaining string**.

Real input example with a breakdown into tokens using regular expressions and how a lexer (lexical analyzer) would process it.

```
int x = 42 + y;
```

1. **Lexer** starts at the beginning:
   - Matches "int" using keyword RegEx.
   - Skips the whitespace (whitespace is usually ignored or handled separately).
   - Matches "x" as an identifier.
   - Matches "=" as assignment operator.
   - Matches "42" as a numeric constant.
   - Matches "+" as operator.
   - Matches "y" as another identifier.
   - Matches ";" as a delimiter.

2. The output of the lexer will be something like

<KEYWORD,int> <ID,x> <ASSIGN,=> <NUM,42>
<PLUS,+> <ID,y> <SEMI,;>

Key Concepts Applied

- Regular expressions were used to define each token type.
- The lexer scanned the input left to right.
- It matched the **longest possible prefix** for each token type.
- Tokens are passed to the **parser** for syntax analysis.

How to Handle Spaces and Comments?

- We could create a token Whitespace

  Whitespace = (‘ ’ + ‘\n’ + ‘\t’)+

  - We could also add comments in there
  - An input “ \t\n 5555 “ is transformed into Whitespace Integer Whitespace

- Lexer 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

Why This Matters

- **Lexers** ignore spaces when scanning for tokens.
- **Parsers** are not concerned with spaces — only with the stream of tokens.

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
  - $x_1 \ldots x_K \in L(R)$

Ex: if R represents identifiers, both $x_1 \ldots x_i$ and $x_1 \ldots x_k$ could be valid identifiers, and the lexer is unsure **where to stop**.

- Rule: Pick the longest possible substring, the "maximal munch"

Ambiguities (2)

Which token is used? What if

- $x_1 \ldots x_i \in L(R_j)$ and also
- $x_1 \ldots x_i \in L(R_k)$

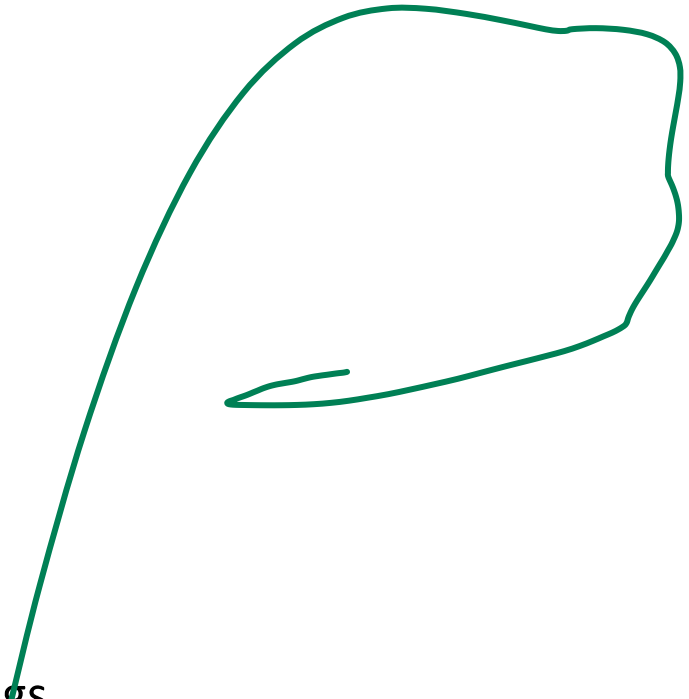
Example:

- R_1 = Keyword and R_2 = Identifier
- "if" matches both, treats "if" as a keyword not an identifier

Rule: use rule listed first (**j if j < k**)

(1) and (2) illustrate **ambiguity in lexical analysis**—specifically, ambiguity during **token recognition** from a stream of characters.

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
- Lexer tools allow the writing of: R = R1 + ... + Rn + 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 class)
  - Require only single pass over the input
  - Few operations per character (table lookup)