Compilers

Lexical Analysis
1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation
if ( i == j )
    Z = 0 ;
else
    Z = 1 ;

\textbf{tif ( i == j )\textbackslash n\textbackslash t\textbackslash tz =0;\textbackslash n\textbackslash telse\textbackslash n\textbackslash t\textbackslash tz =1 ;}

\textbullet Goal: Partition input string into substrings
    \textendash Where the substrings are tokens
Token Class (or Class)

- In English:
  - noun, verb, adjective, etc.

- In a programming language:
  - identifier, keyword, ‘(‘, ‘)’, numbers, etc.
• Token classes 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, 00

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

• **Whitespace:**
  – *a non-empty sequence of blanks, newlines, and tabs*
Lexical Analysis

- Classify program substrings according to role
- Communicate tokens to the parser
  - Tokens are pairs of classes and strings which are inputs to the parser
  - Foo=42 → <Id, "Foo">, <Op, "=">, <Int, "42">

- Parser relies on token distinctions
  - An identifier is treated differently from a keyword

- Define a finite set of tokens
  - Tokens describe all items of interest
  - Choice of tokens depends on language, design of Parser
tif (i == j) \nt\ntz = 0; \ntelse \nt\ntz = 1;

• Useful tokens for this expression:
  ▪ Integer, Keyword, Operator, Identifier, Whitespace,
  ▪ (, ), =, ;

• (, ), =, ; are tokens, not characters, here
tif (i == j)\n\nt\nctz =0;\n\ntelse\n\nt\nctz = 1 ;
W KW(IWO WI) W IW=WN; W K W IW= WN;

W: Whitespace
K: Keyword
I: Identifier
O: Operation
N: Number
(
)
=
;
For the code fragment below, choose the correct number of tokens in each class that appear in the code fragment

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

- W: Whitespace
- K: Keyword
- I: Identifier
- N: Number
- O: Other Tokens:
  ```plaintext
  = ; ++ > ( ) { }
  ```

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

```plaintext
x = 0;\n\ntwhile (x > 10){\n\n\n}\n```

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

Options:

- 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
• An implementation must do two things:

1. Recognize substrings corresponding to tokens
   • The *lexemes*

2. Identify the token class of each lexeme
• FORTRAN rule: Whitespace is insignificant

• \texttt{VAR1} \textit{is the same as} \texttt{VAR1}  
  \begin{itemize}
  \item on punch card machines it was easy to add extra blanks by accident
  \end{itemize}

• A terrible design!
DO 5 I = 1,25

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

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

DO 5 I = 1.25

Lookahead
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.

We would like to minimize Lookahead, ideally bound it to some constant.
if (i == j)
    Z=0;
else
    Z=1;
if \( i == j \)
\[ Z = 0; \]

else
\[ Z = 1; \]
PL/I keywords are not reserved
PL/I: Programming Language 1, designed by IBM

IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
DECLARE (ARG1, ..., ARGN)

Is DECLARE is a keyword or an array reference?

Unbounded Lookahead
• C++ template syntax:
  
  ```
  Foo<Bar>
  ```

• C++ stream syntax:

  ```
  cin >> var;
  ```

  ```
  foo<bar<bazz>>
  ```
• 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
Compilers

Regular Languages
• There are several formalisms for specifying tokens

• Regular languages are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations
Def. Let $\Sigma$ be a set of characters (an *alphabet*). A *language over* $\Sigma$ is a set of strings of characters drawn from $\Sigma$. 
Example of Languages

- Alphabet = English characters
- Language = English sentences
- Not every string of English characters is an sentence

- Alphabet = ASCII
- Language = C programs

- Note: ASCII character set is different from English character set
• 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**
  
  \[ \epsilon = \{""\} \]
Compound Regular Expressions

- **Union**

\[ A + B = \{a | a \in A\} \cup \{b | b \in B\} \]

- **Concatenation**

\[ AB = \{ab | a \in A \land b \in B\} \]

- **Iteration**

\[ A^* = \bigcup_{i \geq 0} A^i \quad A^i = AA \ldots A \text{ \(i\) times} \]

\[ A^0 = \{\epsilon\} \]
• **Def.** The *regular expressions over* $\Sigma$ *are the smallest set of expressions including*

$$\varepsilon$$

'c' where $c \in \Sigma$

$A + B$ where $A$ and $B$ are expr over $\Sigma$

$AB$ where $A$ and $B$ are expr over $\Sigma$

$A^*$ where $A$ is expr over $\Sigma$
Choose the regular languages that are equivalent to the given regular language: \( (0+1)^*1(0+1) \)

- \((01 + 11)^*(0 + 1)^*\)
- \((0 + 1)^*(10 + 11 + 1)(0 + 1)\)
- \((1 + 0)^*1(1 + 0)\)
- \((0 + 1)^*(0 + 1)(0 + 1)^*\)

\[ \Sigma = \{0, 1\} \]
Choose the regular languages that are equivalent to the given regular language: \((0+1)^*1(0+1)\)

- \((01 + 11)^*(0 + 1)^*\)
- \((0 + 1)^*(10 + 11 + 1)(0 + 1)\)
- \((1 + 0)^*1(1 + 0)\)
- \((0 + 1)^*(0 + 1)(0 + 1)^*\)

\[\Sigma = \{0, 1\}\] Both are correct
• Regular expressions specify regular languages

• Five constructs
  – Two base cases
    • empty and 1-character strings
  – Three compound expressions
    • union, concatenation, iteration
• To be careful, we should distinguish syntax and semantics

• Meaning function $L$ maps syntax to semantics
  • $L : \text{Expressions} \rightarrow \text{Sets of Strings}$

\[
\begin{align*}
\varepsilon &= \{"\}\n\\
'c' &= \{"c"\}
\\
A + B &= A \cup B
\\
AB &= \{ab \mid a \in A \land b \in B\}
\\
A^* &= \bigcup_{i \geq 0} A^i
\end{align*}
\]
To be careful, we should distinguish syntax and semantics

Meaning function $L$ maps syntax to semantics

- $L : \text{Expressions} \rightarrow \text{Sets of Strings}$

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

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

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

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

$L(A^*) = \bigcup_{i \geq 0} L(A^i)$
To be careful, we should distinguish syntax and semantics

Meaning function $L$ maps syntax to semantics

$L : \text{Expressions} \rightarrow \text{Sets of Strings}$

- $L(\varepsilon) = \{"\}\$
- $L(\text{'c'}) = \{\text{'c'}\}\$
- $L(A + B) = L(A) \cup L(B)$
- $L(AB) = L(A)L(B)$
- $L(A^*) = L(A)^*$
• 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

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r + s = s + r$</td>
<td>$+$ is commutative</td>
</tr>
<tr>
<td>$r + (s + t) = (r + s) + t$</td>
<td>$+$ is associative</td>
</tr>
<tr>
<td>$(r \cdot s) \cdot t = r \cdot (s \cdot t)$</td>
<td>concatenation is associative</td>
</tr>
<tr>
<td>$r (s + t) = rs + rt$</td>
<td>concatenation distributes over $+$</td>
</tr>
<tr>
<td>$(s + t) r = s r + t r$</td>
<td></td>
</tr>
<tr>
<td>$\varepsilon r = r$</td>
<td>$\varepsilon$ is the identity element for concatenation</td>
</tr>
<tr>
<td>$r \varepsilon = r$</td>
<td></td>
</tr>
<tr>
<td>$r^* = (r + \varepsilon)^*$</td>
<td>relation between $*$ and $\varepsilon$</td>
</tr>
<tr>
<td>$r^{**} = r^*$</td>
<td>$*$ is idempotent</td>
</tr>
</tbody>
</table>
Compilers

Lexical Specifications
Keyword: “if” or “else” or “then” or...

‘else’ + ‘if’ + ‘then’ + . . .

Note: ‘else’ abbreviates:
‘e’ ‘l’ ‘s’ ‘e’
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*

\[
\text{letter} = 'A' + \ldots + 'Z' + 'a' + \ldots + 'z' \quad \text{[A-Z a-z]}
\]

\[
\text{identifier} = \text{letter} \ (\text{letter} + \text{digit})^* 
\]
Whitespace: a non-empty sequence of blanks, newlines, and tabs

\((\,\, + \,\n + \,\t)^+)\)
Example: Email Addresses

`anyone@cs.stanford.edu`

\[ \Sigma = \text{letters} \cup \{. , @\} \]

`name = \text{letters}^+`

`address = name'@'name'.'name'.'name`
PASCAL numbers

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

opt_fraction = ( '.'.digits) + ε ('.' digits)? 

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

num = digits opt_fraction opt_exponent
Choose the regular languages that are correct specifications of the English-language description given below:

*Twelve-hour times of the form “04:13PM”. Minutes should always be a two digit number, but hours may be a single digit.*

- $(0 + 1)?[0-9]:(0-5)[0-9](AM + PM)$
- $((0 + \varepsilon)[0-9] + 1[0-2]):[0-5][0-9](AM + PM)$
- $(0*[0-9] + 1[0-2]):[0-5][0-9](AM + PM)$
- $(0?[0-9] + 1(0 + 1 + 2)):[0-5][0-9](AM + PM)$
Choose the regular languages that are correct specifications of the English-language description given below:

Twelve-hour times of the form “04:13PM”. Minutes should always be a two digit number, but hours may be a single digit.

- $(0 + 1)?[0-9] : [0-5][0-9](AM + PM)$
- $((0 + \varepsilon)[0-9] + 1[0-2]) : [0-5][0-9](AM + PM)$
- $(0*[0-9] + 1[0-2]) : [0-5][0-9](AM + PM)$
- $(0?[0-9] + 1(0 + 1 + 2)) : [0-5][0-9](A + P)M$
• Regular expressions describe many useful languages

• Regular languages are a language specification
  – We still need an implementation

• Next time: Given a string $s$ and a rexp $R$, is

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