Compilers

Lexical Analysis
1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation
if ( i == j )
    Z = 0 ;
else
    Z = 1 ;
\tif ( i == j )\n\t\tz =0;\n\t\n\t\ntelse\n\t\tz =1 ;

• Goal: Partition input string into substrings
  – Where the substrings are tokens
• **Token Class** (or Class)
  
  – In English:
    - **noun, verb, adjective, etc.**
  
  – In a programming language:
    - **identifier, keyword, `(`, `)`, numbers, etc.**
Lexical Analysis

- 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) t\tz =0; telse t\tz=1;

• Useful tokens for this expression:
  - Integer, Keyword, Operator, Identifier, Whitespace,
  - (, ), =, ;
• (, ), =, ; are tokens, not characters, here
tif (i == j) \nttz = 0; \ntelse \nttz = 1;

KW(IWO WI) IW=WN; 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;
while (x > 10)
{
    x++; 
}
```

- **W**: Whitespace
- **K**: Keyword
- **I**: Identifier
- **N**: Number
- **O**: Other Tokens

Options:
1. W = 9; K = 1; I = 3; N = 2; O = 9
2. W = 11; K = 4; I = 0; N = 2; O = 9
3. W = 9; K = 4; I = 0; N = 3; O = 9
4. 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

```c
x = 0; \n \t while (x > 10) { \n \txt++; \n }
```

- **W**: Whitespace
- **K**: Keyword
- **I**: Identifier
- **N**: Number
- **O**: Other Tokens

- **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

• VAR1 is the same as VA R1
  ▪ on punch card machines it was easy to add extra blanks by accident

• 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

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;
\[
\begin{align*}
\text{if } (i &\leq j) \\
Z &= 0; \\
\text{else} \\
Z &= 1;
\end{align*}
\]
PL/I keywords are not reserved
PL/I: Programming Language 1, designed by IBM

IF ELSE THEN THEN = ELSE; 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

Syntax
CodeGen
Semantics
Types
• 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

    \[ \varepsilon = \{ "" \} \]
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**

\[ A^* = \bigcup_{i \geq 0} A^i \]

\[ A^i = AA \ldots A \text{  i times} \]

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

\[ e \]

\[ 'c' \text{ where } c \in \Sigma \]

\[ A + B \text{ where } A \text{ and } B \text{ are expr over } \Sigma \]

\[ AB \text{ where } A \text{ and } B \text{ are expr over } \Sigma \]

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

\[
\begin{align*}
\square & \quad (01 + 11)^*(0 + 1)^* \\
\square & \quad (0 + 1)^*(10 + 11 + 1)(0 + 1) \\
\square & \quad (1 + 0)^*1(1 + 0)^* \\
\square & \quad (0 + 1)^*(0 + 1)(0 + 1)^* \\
\end{align*}
\]

\[\Sigma = \{0, 1\}\]
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 Languages

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

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

Meaning function $L$ maps syntax to semantics:

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

### Examples

- $L(\varepsilon) = \{\varepsilon\}$
- $L('c') = \{'c'\}$
- $L(A + B) = L(A) \cup L(B)$
- $L(AB) = \{ab \mid a \in L(A) \land b \in L(B)\}$
- $L(A^*) = \bigcup_{i \geq 0} L(A^i)$
• To be careful, we should distinguish syntax and semantics

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

\[
\begin{align*}
L(\varepsilon) &= \{\varepsilon\} \\
L('c') &= \{\text{"c"}\} \\
L(A + B) &= L(A) \cup L(B) \\
L(AB) &= L(A)L(B) \\
L(A^*) &= L(A)^*
\end{align*}
\]
• 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 \ s) \ t = r \ (s \ t))</td>
<td>concatenation is associative</td>
</tr>
<tr>
<td>( r (s + t) = rs + rt ) ( (s + t) \ r = s \ r + t \ r )</td>
<td>concatenation distributes over +</td>
</tr>
<tr>
<td>( \epsilon r = r ) ( r \epsilon = r )</td>
<td>( \epsilon ) is the identity element for concatenation</td>
</tr>
<tr>
<td>( r^* = (r + \epsilon)^* )</td>
<td>relation between * and ( \epsilon )</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*

\[
\text{letter} = \text{‘A’ + . . . + ‘Z’ + ‘a’ + . . . + ‘z’} \quad \text{[A-Z a-z]}
\]

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

(\'\' + '\n' + '\t')+
Example: Email Addresses

\[ \text{address} = \text{name}@\text{name}.'\text{name}.'\text{name} \]

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

\[ \text{name} = \text{letters}^+ \]
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
um = 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.

☐ $(0 + 1)?[0-9]:[0-5][0-9](AM + PM)$

☐ $((0 + \epsilon)[0-9] + 1[0-2]):[0-5][0-9](AM + PM)$

☐ $(0*[0-9] + 1[0-2]):[0-5][0-9](AM + PM)$

☐ $(0?[0-9] + 1(0 + 1 + 2)): [0-5][0-9](A + P)M$
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

\[ s \in L(R) \]
• Lexical Structure = token classes

• We must say what set of strings is in a token class
  - Use *regular languages*