2. Lexical Analysis

Andrea Polini, Luca Tesei

Formal Languages and Compilers
MSc in Computer Science
University of Camerino

ToC

Lexical Analysis

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

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

Lexical Analysis

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

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

Token, Pattern Lexeme

Token

A token is a pair consisting of a token name and an optional attribute value. The token names are the input symbols that the parser processes.

Pattern

A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.

Lexeme

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Lexical Analysis

- Token Class (or Class)
  - In English: *Noun, Verb, Adjective, Adverb, Article, . . .*
  - In a programming language: *Identifier, Keywords, "(", ")", Numbers, . . .*

Lexical Analysis

- Token classes corresponds to sets of strings
- Identifier
  - strings of letter or digits starting with a letter
- Integer
  - a non-empty string of digits
- Keyword
  - "else", "if", "while", ...
- Whitespace
  - a non-empty sequence of blanks, newlines, and tabs

Lexical Analysis

- Token classes corresponds to sets of strings
- Identifier
  - strings of letter or digits starting with a letter
- Integer
  - a non-empty string of digits
- Keyword
  - “else”, “if”, “while”, . . .
- Whitespace
  - a non-empty sequence of blanks, newlines, and tabs

Lexical Analysis

- Token classes corresponds to sets of strings
- Identifier
  - strings of letter or digits starting with a letter
- Integer
  - a non-empty string of digits
- Keyword
  - "else", "if", "while", ...
- Whitespace
  - a non-empty sequence of blanks, newlines, and tabs

Lexical Analysis

- Token classes corresponds to sets of strings
- Identifier
  - strings of letter or digits starting with a letter
- Integer
  - a non-empty string of digits
- Keyword
  - “else”, “if”, “while”, ...
- Whitespace
  - a non-empty sequence of blanks, newlines, and tabs

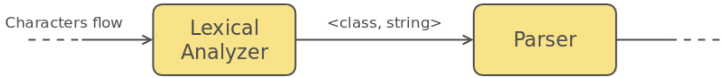
Lexical Analysis

- Token classes corresponds to sets of strings
- Identifier
  - strings of letter or digits starting with a letter
- Integer
  - a non-empty string of digits
- Keyword
  - “else”, “if”, “while”, . . .
- Whitespace
  - a non-empty sequence of blanks, newlines, and tabs

Lexical Analysis

Therefore the role of the lexical analyser (Lexer) is:

- Classify program substring according to role (token class)
- communicate tokens to parser

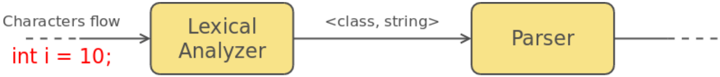


Why is not wise to merge the two components?

Lexical Analysis

Therefore the role of the lexical analyser (Lexer) is:

- Classify program substring according to role (token class)
- communicate tokens to parser

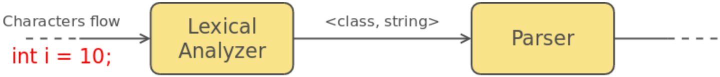


Why is not wise to merge the two components?

Lexical Analysis

Therefore the role of the lexical analyser (Lexer) is:

- Classify program substring according to role (token class)
- communicate tokens to parser



Why is not wise to merge the two components?

Lexical Analysis

Let's analyse these lines of code:

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

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

Token Classes: Identifier, Integer, Keyword, Whitespace

Lexical Analysis

Therefore an implementation of a lexical analyser must do two things:

- Recognise substrings corresponding to tokens
  - the lexemes
- Identify the token class for each lexemes

Lexical Analysis - Tricky problems

- FORTRAN rule: whitespace is insignificant
  - i.e. VA R1 is the same as VAR1

```
DO 5 I = 1,25

DO 5 I = 1.25
```

In FORTRAN the “5” refers to a label you will find in the following of the program code

Lexical Analysis - Tricky problems

- The goal is to partition the string. This is implemented by reading left-to-right, recognising one token at a time
- “Lookahead” may be required to decide where one token ends and the next token begins
- PL/1 keywords are not reserved

```
IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
```

```
DECLARE(ARG1,...,ARGN)
```

Is `DECLARE` a keyword or an array reference?

Need for an unbounded lookahead

Lexical Analysis - Tricky problems

- The goal is to partition the string. This is implemented by reading left-to-right, recognising one token at a time
- “Lookahead” may be required to decide where one token ends and the next token begins
- PL/1 keywords are not reserved

```
IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
```

```
DECLARE(ARG1,...,ARGN)
```

Is `DECLARE` a keyword or an array reference?

Need for an unbounded lookahead

Lexical Analysis - Tricky problems

- The goal is to partition the string. This is implemented by reading left-to-right, recognising one token at a time
- “Lookahead” may be required to decide where one token ends and the next token begins
- PL/1 keywords are not reserved

```
IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
```

```
DECLARE(ARG1,...,ARGN)
```

Is `DECLARE` a keyword or an array reference?

Need for an unbounded lookahead

Lexical Analysis - Tricky problems

- C++ template syntax:

```
Foo<Bar>
```

- C++ stream syntax:

```
cin >> var;
```

```
Foo<Bar<Barr>>
```

Lexical Analysis - Tricky problems

- C++ template syntax:

```
Foo<Bar>
```

- C++ stream syntax:

```
cin >> var;
```

```
Foo<Bar<Barr>>
```

ToC

Languages

Language

Let Σ be a set of characters generally referred to as the *alphabet*. A language over Σ is a set of strings of characters drawn from Σ

Alphabet = English character $\Longrightarrow$ Language = English sentences
Alphabet = ASCII $\Longrightarrow$ Language = C programs

Given $\Sigma = \{a, b\}$ examples of simple languages are:

- $\mathcal{L}_1 = \{a, ab, aa\}$
- $\mathcal{L}_2 = \{b, ab, aabb\}$
- $\mathcal{L}_3 = \{s \mid s$ has an equal number of a's and b's$\}$
- ...

Grammar Definition

Grammar

A Grammar $\mathcal{G}$ is a tuple $\langle \mathcal{V}_\mathcal{T}, \mathcal{V}_\mathcal{N}, \mathcal{S}, \mathcal{P} \rangle$ where:

- $\mathcal{V}_\mathcal{T}$ is a finite and non empty set of terminal symbols (alphabet)
- $\mathcal{V}_\mathcal{N}$ is a finite set of non-terminal symbols s.t. $\mathcal{V}_\mathcal{N} \cap \mathcal{V}_\mathcal{T} = \varnothing$
- $\mathcal{S} \in \mathcal{V}_\mathcal{N}$ is the start symbol
- $\mathcal{P}$ is a finite set of productions s.t. $\mathcal{P} \subseteq (\mathcal{V}^* \cdot \mathcal{V}_\mathcal{N} \cdot \mathcal{V}^*) \times \mathcal{V}^*$ where $\mathcal{V}^* = \mathcal{V}_\mathcal{T} \cup \mathcal{V}_\mathcal{N}$

Derivations

Derivations

Given a grammar $\mathcal{G} = \langle \mathcal{V}_\mathcal{T}, \mathcal{V}_\mathcal{N}, \mathcal{S}, \mathcal{P} \rangle$ a derivation is a sequence of strings $\phi_1, \phi_2, ..., \phi_n$ s.t.
$\forall i \in \{1, .., n\}.\phi_i \in \mathcal{V}^* \wedge \forall i \in \{1, ..., n-1\}.\exists p \in \mathcal{P} : \phi_i \rightarrow^p \phi_{i+1}$
We generally write $\phi_1 \rightarrow^* \phi_n$ to indicate that from ϕ_1 it is possible to derive ϕ_n repeatedly applying productions in $\mathcal{P}$

Generated Language

The language generated by a grammar $\mathcal{G} = \langle \mathcal{V}_\mathcal{T}, \mathcal{V}_\mathcal{N}, \mathcal{S}, \mathcal{P} \rangle$ corresponds to: $\mathcal{L}(\mathcal{G}) = \{x \mid x \in \mathcal{V}_T^* \wedge \mathcal{S} \rightarrow^* x\}$

Derivations

Derivations

Given a grammar $\mathcal{G} = \langle \mathcal{V}_\mathcal{T}, \mathcal{V}_\mathcal{N}, \mathcal{S}, \mathcal{P} \rangle$ a derivation is a sequence of strings $\phi_1, \phi_2, ..., \phi_n$ s.t.
$\forall i \in \{1, .., n\}.\phi_i \in \mathcal{V}^* \wedge \forall i \in \{1, ..., n-1\}.\exists p \in \mathcal{P} : \phi_i \rightarrow^p \phi_{i+1}$
We generally write $\phi_1 \rightarrow^* \phi_n$ to indicate that from ϕ_1 it is possible to derive ϕ_n repeatedly applying productions in $\mathcal{P}$

Generated Language

The language generated by a grammar $\mathcal{G} = \langle \mathcal{V}_\mathcal{T}, \mathcal{V}_\mathcal{N}, \mathcal{S}, \mathcal{P} \rangle$ corresponds to: $\mathcal{L}(\mathcal{G}) = \{x \mid x \in \mathcal{V}_T^* \wedge \mathcal{S} \rightarrow^* x\}$

Chomsky Hierarchy

A hierarchy of grammars can be defined imposing constraints on the structure of the productions in set $\mathcal{P}$ ($\alpha, \beta, \gamma \in \mathcal{V}^*, a \in \mathcal{V}_T, A, B \in \mathcal{V}_N$):

T0. Unrestricted Grammars:
- Production Schema: *no constraints*
- Recognizing Automaton: Turing Machines

T1. Context Sensitive Grammars:
- Production Schema: $\alpha A\beta \rightarrow \alpha\gamma\beta$
- Recognizing Automaton: Linear Bound Automaton (LBA)

T2. Context-Free Grammars:
- Production Schema: $A \rightarrow \gamma$
- Recognizing Automaton: Non-deterministic Push-down Automaton

T3. Regular Grammars:
- Production Schema: $A \rightarrow a$ or $A \rightarrow aB$
- Recognizing Automaton: Finite State Automaton

Meaning function $\mathscr{L}$

Meaning Function

Once you defined a way to describe the strings in a language it is important to define a meaning function $\mathscr{L}$ that maps syntax to semantics

- e.g. the case for numbers

- Why using a meaning function?
  - Makes clear what is syntax, what is semantics
  - Allows us to consider notation as a separate issue
  - Expressions and meanings are not 1 to 1

Warning

It should never happen that the same syntactical structure has more meanings

Meaning function $\mathscr{L}$

Meaning Function

Once you defined a way to describe the strings in a language it is important to define a meaning function $\mathscr{L}$ that maps syntax to semantics

- e.g. the case for numbers

- Why using a meaning function?
  - Makes clear what is syntax, what is semantics
  - Allows us to consider notation as a separate issue
  - Expressions and meanings are not 1 to 1

Warning

It should never happen that the same syntactical structure has more meanings

Meaning function $\mathscr{L}$

Meaning Function

Once you defined a way to describe the strings in a language it is important to define a meaning function $\mathscr{L}$ that maps syntax to semantics

- e.g. the case for numbers

- Why using a meaning function?
  - Makes clear what is syntax, what is semantics
  - Allows us to consider notation as a separate issue
  - Expressions and meanings are not 1 to 1

Warning

It should never happen that the same syntactical structure has more meanings