Lexical Analysis

Lecture 2

Objectives

By the end of this lecture you should be able to:

1. Identify the role of lexical analysis in a compiler.
2. Design regular definitions for regular languages.
3. Design action-augmented regular definitions for regular languages.
4. Design fallback DFA with actions for regular languages.

Outline

Outline

What It Does

Main Function

1. Partition the input stream into lexemes.
2. Generate a token for each lexeme.

Auxiliary Function

Ignores substrings which are insignificant for the compiling process.

- e.g., comments and white spaces.

Other Possible Functions

- Macro expansion.
- Keeping track of line numbers for informative error messages.

What It Does

Main Function

1. Partition the input stream into lexemes.
2. Generate a token for each lexeme.

Auxiliary Function

Ignores substrings which are insignificant for the compiling process.

- e.g., comments and white spaces.

Other Possible Functions

- Macro expansion.
- Keeping track of line numbers for informative error messages.

What It Does

Main Function

1. Partition the input stream into lexemes.
2. Generate a token for each lexeme.

Auxiliary Function

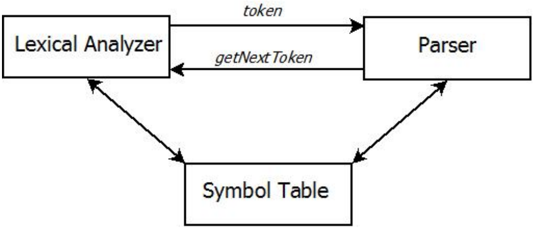
Ignores substrings which are insignificant for the compiling process.

- e.g., comments and white spaces.

Other Possible Functions

- Macro expansion.
- Keeping track of line numbers for informative error messages.

Connection to the Rest of the System



Associated Concepts: Tokens

Tokens

- A token is a tuple of the form $\langle L, A\rangle$ or $\langle L\rangle$, where
  - L is the name of a lexical category, and
  - A is an attribute.
- Lexical categories are terminals from the parser perspective.
- Attributes carry semantically-relevant information about the particular instance of L encountered.
- No attributes are needed if L is a single-instance category. (Hence, the second form)
- An instance of a lexical category is called a lexeme.

Associated Concepts: Tokens

Tokens

- A token is a tuple of the form $\langle L, A \rangle$ or $\langle L \rangle$, where
  - L is the name of a lexical category, and
  - A is an attribute.
- Lexical categories are terminals from the parser perspective.
- Attributes carry semantically-relevant information about the particular instance of L encountered.
- No attributes are needed if L is a single-instance category. (Hence, the second form)
- An instance of a lexical category is called a lexeme.

Associated Concepts: Tokens

Tokens

- A token is a tuple of the form $\langle L, A\rangle$ or $\langle L\rangle$, where
  - L is the name of a lexical category, and
  - A is an attribute.
- Lexical categories are terminals from the parser perspective.
- Attributes carry semantically-relevant information about the particular instance of L encountered.
- No attributes are needed if L is a single-instance category. (Hence, the second form)
- An instance of a lexical category is called a lexeme.

Associated Concepts: Tokens

Tokens

- A token is a tuple of the form $\langle L, A\rangle$ or $\langle L\rangle$, where
  - L is the name of a lexical category, and
  - A is an attribute.
- Lexical categories are terminals from the parser perspective.
- Attributes carry semantically-relevant information about the particular instance of L encountered.
- No attributes are needed if L is a single-instance category. (Hence, the second form)
- An instance of a lexical category is called a lexeme.

Associated Concepts: Tokens

Tokens

- A token is a tuple of the form $\langle L, A \rangle$ or $\langle L \rangle$, where
  - L is the name of a lexical category, and
  - A is an attribute.
- Lexical categories are terminals from the parser perspective.
- Attributes carry semantically-relevant information about the particular instance of L encountered.
- No attributes are needed if L is a single-instance category. (Hence, the second form)
- An instance of a lexical category is called a lexeme.

Associated Concepts: Tokens

Tokens

- A token is a tuple of the form $\langle L, A\rangle$ or $\langle L\rangle$, where
  - L is the name of a lexical category, and
  - A is an attribute.
- Lexical categories are terminals from the parser perspective.
- Attributes carry semantically-relevant information about the particular instance of L encountered.
- No attributes are needed if L is a single-instance category. (Hence, the second form)
- An instance of a lexical category is called a lexeme.

Associated Concepts: Tokens

Tokens

- A token is a tuple of the form $\langle L, A\rangle$ or $\langle L\rangle$, where
  - L is the name of a lexical category, and
  - A is an attribute.
- Lexical categories are terminals from the parser perspective.
- Attributes carry semantically-relevant information about the particular instance of L encountered.
- No attributes are needed if L is a single-instance category. (Hence, the second form)
- An instance of a lexical category is called a lexeme.

Associated Concepts: Patterns

Patterns

- The set of instances of a lexical category constitute a language.
- Typically, this language is regular.
- A pattern for a particular lexical category is a description of the form of lexemes in that language.
- This description is sometimes a simple enumeration of the lexemes.
  - As in natural languages.
- May also be described by some kind of grammar; typically a regular grammar or, equivalently, a regular expression.

Associated Concepts: Patterns

Patterns

- The set of instances of a lexical category constitute a language.
- Typically, this language is regular.
- A pattern for a particular lexical category is a description of the form of lexemes in that language.
- This description is sometimes a simple enumeration of the lexemes.
  - As in natural languages.
- May also be described by some kind of grammar; typically a regular grammar or, equivalently, a regular expression.

Associated Concepts: Patterns

Patterns

- The set of instances of a lexical category constitute a language.
- Typically, this language is regular.
- A pattern for a particular lexical category is a description of the form of lexemes in that language.
- This description is sometimes a simple enumeration of the lexemes.
  - As in natural languages.
- May also be described by some kind of grammar; typically a regular grammar or, equivalently, a regular expression.

Associated Concepts: Patterns

Patterns

- The set of instances of a lexical category constitute a language.
- Typically, this language is regular.
- A pattern for a particular lexical category is a description of the form of lexemes in that language.
- This description is sometimes a simple enumeration of the lexemes.
  - As in natural languages.
- May also be described by some kind of grammar; typically a regular grammar or, equivalently, a regular expression.

Associated Concepts: Patterns

Patterns

- The set of instances of a lexical category constitute a language.
- Typically, this language is regular.
- A pattern for a particular lexical category is a description of the form of lexemes in that language.
- This description is sometimes a simple enumeration of the lexemes.
  - As in natural languages.
- May also be described by some kind of grammar; typically a regular grammar or, equivalently, a regular expression.

Associated Concepts: Patterns

Patterns

- The set of instances of a lexical category constitute a language.
- Typically, this language is regular.
- A pattern for a particular lexical category is a description of the form of lexemes in that language.
- This description is sometimes a simple enumeration of the lexemes.
  - As in natural languages.
- May also be described by some kind of grammar; typically a regular grammar or, equivalently, a regular expression.

Example

Example

Lexical Category	Pattern
if	{`if`, `If`, `iF`, `IF`}
else	{`else`, ..., `ELSE`}
comp	$\{>, <, >=, <=, ==, != \}$
id	any letter followed by letters or digits
num	any numeric literal
lit	any string between " and "
lp	{(}
rp	{)}

- Input: `if (x > 10) printf("Yes") else printf("No")`
- Output:
  [⟨**if**⟩, ⟨**lp**⟩, ⟨**id**, 1⟩, ⟨**comp**, >⟩, ⟨**num**, 10⟩, ⟨**rp**⟩,
  ⟨**id**, 2⟩, ⟨**lp**⟩, ⟨**lit**, `Yes`⟩, ⟨**rp**⟩,
  ⟨**else**⟩, ⟨**id**, 2⟩, ⟨**lp**⟩, ⟨**lit**, `No`⟩, ⟨**rp**⟩]

Outline

Regular Expressions

Definition

R is a regular expression over alphabet Σ if R is

1. a for some $a \in \Sigma$,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

- Note:
  - $L(a) = \{a\}$; $L(\varepsilon) = \{\varepsilon\}$; $L(R^*) = (L(R))^*$.
  - $L(R_1 \otimes R_2) = L(R_1) \otimes L(R_2)$, for $\otimes \in \{\cup, \circ\}$.

Regular Expressions

Definition

R is a regular expression over alphabet Σ if R is

1. a for some $a \in \Sigma$,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

- Note:
  - $L(a) = \{a\}$; $L(\varepsilon) = \{\varepsilon\}$; $L(R^*) = (L(R))^*$.
  - $L(R_1 \otimes R_2) = L(R_1) \otimes L(R_2)$, for $\otimes \in \{\cup, \circ\}$.

Extended Language of Regular Expressions

Some convenient abbreviations:

- $R_1|R_2 = R_1 \cup R_2$.
- $R_1R_2 = R_1 \circ R_2$.
- $R^+ = R \circ R^*$.
- $\Sigma = a_1|a_2|\ldots|a_n$, where $\Sigma = \{a_1, a_2, \ldots, a_n\}$.
- $R? = R|\varepsilon$.
- $[a_1a_2\ldots a_n] = a_1|a_2|\ldots|a_n$, where $\{a_1, a_2, \ldots, a_n\} \subseteq \Sigma$.
- $[a_1 - a_n] = [a_1, a_2, \ldots, a_n]$, provided that $\langle a_1, a_2, \ldots, a_n\rangle$ is a natural permutation of $\{a_1, a_2, \ldots, a_n\}$.

Extended Language of Regular Expressions

Some convenient abbreviations:

- $R_1|R_2 = R_1 \cup R_2$.
- $R_1R_2 = R_1 \circ R_2$.
- $R^+ = R \circ R^*$.
- $\Sigma = a_1|a_2|\ldots|a_n$, where $\Sigma = \{a_1, a_2, \ldots, a_n\}$.
- $R? = R|\varepsilon$.
- $[a_1a_2\ldots a_n] = a_1|a_2|\ldots|a_n$, where $\{a_1, a_2, \ldots, a_n\} \subseteq \Sigma$.
- $[a_1 - a_n] = [a_1, a_2, \ldots, a_n]$, provided that $\langle a_1, a_2, \ldots, a_n\rangle$ is a natural permutation of $\{a_1, a_2, \ldots, a_n\}$.

Extended Language of Regular Expressions

Some convenient abbreviations:

- $R_1|R_2 = R_1 \cup R_2$.
- $R_1R_2 = R_1 \circ R_2$.
- $R^+ = R \circ R^*$.
- $\Sigma = a_1|a_2|\ldots|a_n$, where $\Sigma = \{a_1, a_2, \ldots, a_n\}$.
- $R? = R|\varepsilon$.
- $[a_1a_2\ldots a_n] = a_1|a_2|\ldots|a_n$, where $\{a_1, a_2, \ldots, a_n\} \subseteq \Sigma$.
- $[a_1 - a_n] = [a_1, a_2, \ldots, a_n]$, provided that $\langle a_1, a_2, \ldots, a_n\rangle$ is a natural permutation of $\{a_1, a_2, \ldots, a_n\}$.

Extended Language of Regular Expressions

Some convenient abbreviations:

- $R_1|R_2 = R_1 \cup R_2$.
- $R_1R_2 = R_1 \circ R_2$.
- $R^+ = R \circ R^*$.
- $\Sigma = a_1|a_2|\ldots|a_n$, where $\Sigma = \{a_1, a_2, \ldots, a_n\}$.
- $R? = R|\varepsilon$.
- $[a_1a_2\ldots a_n] = a_1|a_2|\ldots|a_n$, where $\{a_1, a_2, \ldots, a_n\} \subseteq \Sigma$.
- $[a_1 - a_n] = [a_1, a_2, \ldots, a_n]$, provided that $\langle a_1, a_2, \ldots, a_n\rangle$ is a natural permutation of $\{a_1, a_2, \ldots, a_n\}$.

Extended Language of Regular Expressions

Some convenient abbreviations:

- $R_1|R_2 = R_1 \cup R_2$.
- $R_1R_2 = R_1 \circ R_2$.
- $R^+ = R \circ R^*$.
- $\Sigma = a_1|a_2|\ldots|a_n$, where $\Sigma = \{a_1, a_2, \ldots, a_n\}$.
- $R? = R|\varepsilon$.
- $[a_1a_2\ldots a_n] = a_1|a_2|\ldots|a_n$, where $\{a_1, a_2, \ldots, a_n\} \subseteq \Sigma$.
- $[a_1 - a_n] = [a_1, a_2, \ldots, a_n]$, provided that $\langle a_1, a_2, \ldots, a_n\rangle$ is a natural permutation of $\{a_1, a_2, \ldots, a_n\}$.

Extended Language of Regular Expressions

Some convenient abbreviations:

- $R_1|R_2 = R_1 \cup R_2$.
- $R_1R_2 = R_1 \circ R_2$.
- $R^+ = R \circ R^*$.
- $\Sigma = a_1|a_2|\dots|a_n$, where $\Sigma = \{a_1, a_2, \dots, a_n\}$.
- $R? = R|\varepsilon$.
- $[a_1a_2\dots a_n] = a_1|a_2|\dots|a_n$, where $\{a_1, a_2, \dots, a_n\} \subseteq \Sigma$.
- $[a_1 - a_n] = [a_1, a_2, \dots, a_n]$, provided that $\langle a_1, a_2, \dots, a_n\rangle$ is a natural permutation of $\{a_1, a_2, \dots, a_n\}$.

Extended Language of Regular Expressions

Some convenient abbreviations:

- $R_1|R_2 = R_1 \cup R_2$.
- $R_1R_2 = R_1 \circ R_2$.
- $R^+ = R \circ R^*$.
- $\Sigma = a_1|a_2|\ldots|a_n$, where $\Sigma = \{a_1, a_2, \ldots, a_n\}$.
- $R? = R|\varepsilon$.
- $[a_1a_2\ldots a_n] = a_1|a_2|\ldots|a_n$, where $\{a_1, a_2, \ldots, a_n\} \subseteq \Sigma$.
- $[a_1 - a_n] = [a_1, a_2, \ldots, a_n]$, provided that $\langle a_1, a_2, \ldots, a_n\rangle$ is a natural permutation of $\{a_1, a_2, \ldots, a_n\}$.

Regular Definition

Definition

A regular definition $\mathfrak{R}(\Sigma, D)$ over alphabets Σ and D is a finite sequence $\langle P_1, \ldots, P_n \rangle$ of pairs $P_i = \langle D_i, R_i \rangle$, where

- $D = \{D_1, \ldots, D_n\}$.
- $D \cap \Sigma = \varnothing$.
- $|D| = n$.
- R_i is a regular expression over $\Sigma \cup \{D_1, \ldots, D_{i-1}\}$, for $1 \leq i \leq n$.

Note:

- D_i is a (user-defined) shorthand for a regular expression over Σ.

Regular Definition

Definition

A regular definition $\mathfrak{R}(\Sigma, D)$ over alphabets Σ and D is a finite sequence $\langle P_1, \ldots, P_n \rangle$ of pairs $P_i = \langle D_i, R_i \rangle$, where

- $D = \{D_1, \ldots, D_n\}$.
- $D \cap \Sigma = \varnothing$.
- $|D| = n$.
- R_i is a regular expression over $\Sigma \cup \{D_1, \ldots, D_{i-1}\}$, for $1 \leq i \leq n$.

Note:

- D_i is a (user-defined) shorthand for a regular expression over Σ.

Regular Definition

Definition

A regular definition $\mathfrak{R}(\Sigma, D)$ over alphabets Σ and D is a finite sequence $\langle P_1, \ldots, P_n \rangle$ of pairs $P_i = \langle D_i, R_i \rangle$, where

- $D = \{D_1, \ldots, D_n\}$.
- $D \cap \Sigma = \varnothing$.
- $|D| = n$.
- R_i is a regular expression over $\Sigma \cup \{D_1, \ldots, D_{i-1}\}$, for $1 \leq i \leq n$.

Note:

- D_i is a (user-defined) shorthand for a regular expression over Σ.

Regular Definition

Definition

A regular definition $\mathfrak{R}(\Sigma, D)$ over alphabets Σ and D is a finite sequence $\langle P_1, \ldots, P_n \rangle$ of pairs $P_i = \langle D_i, R_i \rangle$, where

- $D = \{D_1, \ldots, D_n\}$.
- $D \cap \Sigma = \varnothing$.
- $|D| = n$.
- R_i is a regular expression over $\Sigma \cup \{D_1, \ldots, D_{i-1}\}$, for $1 \leq i \leq n$.

Note:

- D_i is a (user-defined) shorthand for a regular expression over Σ.

Regular Definition

Definition

A regular definition $\mathfrak{R}(\Sigma, D)$ over alphabets Σ and D is a finite sequence $\langle P_1, \ldots, P_n \rangle$ of pairs $P_i = \langle D_i, R_i \rangle$, where

- $D = \{D_1, \ldots, D_n\}$.
- $D \cap \Sigma = \varnothing$.
- $|D| = n$.
- R_i is a regular expression over $\Sigma \cup \{D_1, \ldots, D_{i-1}\}$, for $1 \leq i \leq n$.

Note:

- D_i is a (user-defined) shorthand for a regular expression over Σ.

Regular Definition

Definition

A regular definition $\mathfrak{R}(\Sigma, D)$ over alphabets Σ and D is a finite sequence $\langle P_1, \ldots, P_n \rangle$ of pairs $P_i = \langle D_i, R_i \rangle$, where

- $D = \{D_1, \ldots, D_n\}$.
- $D \cap \Sigma = \varnothing$.
- $|D| = n$.
- R_i is a regular expression over $\Sigma \cup \{D_1, \ldots, D_{i-1}\}$, for $1 \leq i \leq n$.

Note:

- D_i is a (user-defined) shorthand for a regular expression over Σ.

Example 1

Example (C Identifiers)

$$\begin{array}{lcl} letter\_ & \longrightarrow & [A-Za-z] \mid \_ \\ digit & \longrightarrow & [0-9] \\ id & \longrightarrow & letter\_(letter\_|digit)^* \end{array}$$

Example 2

Example (Unsigned Numeric Literals)

$$
\begin{array}{lcl}
\textit{digit} & \longrightarrow & [0-9] \\
\textit{digits} & \longrightarrow & \textit{digit}^{+} \\
\textit{opFrac} & \longrightarrow & (.\ \textit{digits})? \\
\textit{opExp} & \longrightarrow & (\texttt{E}\ [+-]?\ \textit{digits})? \\
\textit{number} & \longrightarrow & \textit{digits opFrac opExp}
\end{array}
$$

How to Build a Lexical Analyzer in Five Steps

1. Write a regular definition where, for each of the n lexical categories of our language L, there is a pair $\langle D_i, R_i \rangle$, where D_i is the category name and R_i is its pattern.
   - That this can be done is a non-trivial, but practically-valid, assumption.
2. For each D_i $(1 \leq i \leq n)$ compile a corresponding regular expression R_i^{Σ} over Σ.
   - How?
3. Construct the expression $R = (R_1^{\Sigma} | R_2^{\Sigma} | \cdots | R_n^{\Sigma})^+$.
   - $L(R) \supseteq L$ is the set of *lexically-correct* strings of our language.
4. Convert R into an equivalent NFA.
5. Convert the NFA into an equivalent DFA.

How to Build a Lexical Analyzer in Five Steps

1. Write a regular definition where, for each of the n lexical categories of our language L, there is a pair $\langle D_i, R_i \rangle$, where D_i is the category name and R_i is its pattern.
   - That this can be done is a non-trivial, but practically-valid, assumption.
2. For each D_i $(1 \leq i \leq n)$ compile a corresponding regular expression R_i^{Σ} over Σ.
   - How?
3. Construct the expression $R = (R_1^{\Sigma} | R_2^{\Sigma} | \cdots | R_n^{\Sigma})^+$.
   - $L(R) \supseteq L$ is the set of *lexically-correct* strings of our language.
4. Convert R into an equivalent NFA.
5. Convert the NFA into an equivalent DFA.

How to Build a Lexical Analyzer in Five Steps

1. Write a regular definition where, for each of the n lexical categories of our language L, there is a pair $\langle D_i, R_i \rangle$, where D_i is the category name and R_i is its pattern.
   - That this can be done is a non-trivial, but practically-valid, assumption.
2. For each D_i ($1 \leq i \leq n$) compile a corresponding regular expression R_i^Σ over Σ.
   - How?
3. Construct the expression $R = (R_1^\Sigma | R_2^\Sigma | \cdots | R_n^\Sigma)^+$.
   - $L(R) \supseteq L$ is the set of *lexically-correct* strings of our language.
4. Convert R into an equivalent NFA.
5. Convert the NFA into an equivalent DFA.

How to Build a Lexical Analyzer in Five Steps

1. Write a regular definition where, for each of the n lexical categories of our language L, there is a pair $\langle D_i, R_i \rangle$, where D_i is the category name and R_i is its pattern.
    - That this can be done is a non-trivial, but practically-valid, assumption.
2. For each D_i $(1 \leq i \leq n)$ compile a corresponding regular expression R_i^{Σ} over Σ.
    - How?
3. Construct the expression $R = (R_1^{\Sigma} | R_2^{\Sigma} | \cdots | R_n^{\Sigma})^+$.
    - $L(R) \supseteq L$ is the set of *lexically-correct* strings of our language.
4. Convert R into an equivalent NFA.
5. Convert the NFA into an equivalent DFA.

How to Build a Lexical Analyzer in Five Steps

1. Write a regular definition where, for each of the n lexical categories of our language L, there is a pair $\langle D_i, R_i\rangle$, where D_i is the category name and R_i is its pattern.
   - That this can be done is a non-trivial, but practically-valid, assumption.
2. For each D_i $(1 \leq i \leq n)$ compile a corresponding regular expression R_i^{Σ} over Σ.
   - How?
3. Construct the expression $R = (R_1^{\Sigma} | R_2^{\Sigma} | \cdots | R_n^{\Sigma})^+$.
   - $L(R) \supseteq L$ is the set of *lexically-correct* strings of our language.
4. Convert R into an equivalent NFA.
5. Convert the NFA into an equivalent DFA.

How to Build a Lexical Analyzer in Five Steps

1. Write a regular definition where, for each of the n lexical categories of our language L, there is a pair $\langle D_i, R_i \rangle$, where D_i is the category name and R_i is its pattern.
   - That this can be done is a non-trivial, but practically-valid, assumption.
2. For each D_i ($1 \leq i \leq n$) compile a corresponding regular expression R_i^{Σ} over Σ.
   - How?
3. Construct the expression $R = (R_1^{\Sigma} | R_2^{\Sigma} | \cdots | R_n^{\Sigma})^+$.
   - $L(R) \supseteq L$ is the set of *lexically-correct* strings of our language.
4. Convert R into an equivalent NFA.
5. Convert the NFA into an equivalent DFA.

How to Build a Lexical Analyzer in Five Steps

1. Write a regular definition where, for each of the n lexical categories of our language L, there is a pair $\langle D_i, R_i \rangle$, where D_i is the category name and R_i is its pattern.
   - That this can be done is a non-trivial, but practically-valid, assumption.
2. For each D_i $(1 \leq i \leq n)$ compile a corresponding regular expression R_i^{Σ} over Σ.
   - How?
3. Construct the expression $R = (R_1^{\Sigma} | R_2^{\Sigma} | \cdots | R_n^{\Sigma})^+$.
   - $L(R) \supseteq L$ is the set of *lexically-correct* strings of our language.
4. Convert R into an equivalent NFA.
5. Convert the NFA into an equivalent DFA.

How to Build a Lexical Analyzer in Five Steps

1. Write a regular definition where, for each of the n lexical categories of our language L, there is a pair $\langle D_i, R_i \rangle$, where D_i is the category name and R_i is its pattern.
   - That this can be done is a non-trivial, but practically-valid, assumption.
2. For each D_i $(1 \leq i \leq n)$ compile a corresponding regular expression R_i^{Σ} over Σ.
   - How?
3. Construct the expression $R = (R_1^{\Sigma} | R_2^{\Sigma} | \cdots | R_n^{\Sigma})^+$.
   - $L(R) \supseteq L$ is the set of *lexically-correct* strings of our language.
4. Convert R into an equivalent NFA.
5. Convert the NFA into an equivalent DFA.

But . . .

- This can only *recognize* lexically-correct programs.
- It does not split the input into lexemes and does not generate a stream of tokens.
- Need to augment this procedure with mechanisms to do so.

Outline

Lexical Ambiguity

- When scanning a program from left to right, we do not know where a potential lexeme ends.
- This is caused by two types of *lexical ambiguity*:
    1. There is more than one way to split the program into lexemes.
    2. For a fixed splitting, there is more than one stream of tokens that can be generated.
- The first kind of ambiguity occurs when one lexeme is a proper prefix of another.
- The second occurs when a lexeme matches the patterns of more than one lexical category.

Examples

Example (Artificial Languages)

Let $R = (\mathtt{a} \mid \mathtt{abb} \mid \mathtt{a}^*\mathtt{b}^+)^*$.

1. The string `aabb` could be split as [`a`, `abb`] or as [`aabb`].
2. The string `abb` matches both `abb` and $\mathtt{a}^*\mathtt{b}^+$.

Example (Programming Languages)

- $<=$ is either $[\langle \textbf{comp}, < \rangle, \langle \textbf{comp}, = \rangle]$ or $[\langle \textbf{comp}, <= \rangle]$.
- If keywords are reserved, then `if` is either $[\langle \textbf{if} \rangle]$ or $[\langle \textbf{id}, ? \rangle]$.

Examples

Example (Artificial Languages)

Let $R = (\mathtt{a} \mid \mathtt{abb} \mid \mathtt{a}^*\mathtt{b}^+)^*$.

1. The string $\mathtt{aabb}$ could be split as $[\mathtt{a}, \mathtt{abb}]$ or as $[\mathtt{aabb}]$.
2. The string $\mathtt{abb}$ matches both $\mathtt{abb}$ and $\mathtt{a}^*\mathtt{b}^+$.

Example (Programming Languages)

- $<=$ is either $[\langle \textbf{comp}, < \rangle, \langle \textbf{comp}, = \rangle]$ or $[\langle \textbf{comp}, <= \rangle]$.
- If keywords are reserved, then `if` is either $[\langle \textbf{if} \rangle]$ or $[\langle \textbf{id}, ? \rangle]$.

Common Disambiguation Strategies

- For the first type of ambiguity, opt for the splitting with the longest possible prefix lexeme:
  - If $[l_{11}, l_{21}, \ldots, l_{n1}]$ and $[l_{12}, l_{22}, \ldots, l_{m2}]$ are two splittings with $|l_{i1}| > |l_{i2}|$ and $l_{j1} = l_{j2}$ for $1 \leq j < i$, choose the first.
- For the second type, opt for the lexical category whose pattern appears earlier in the regular definition.
  - If the lexeme matches p_i and p_j, where $i < j$, choose $\langle D_i, p_i \rangle$.

Outline

Action-Augmented Regular Definitions

Definition

An action-augmented regular definition is a triple $\langle \mathfrak{R}(\Sigma, D), \mathcal{C}, \mathcal{A} \rangle$, where

- $\mathfrak{R}$ is a regular definition,
- $\mathcal{C} \subseteq D$, and
- $\mathcal{A}$ is a function which maps every $c \in \mathcal{C}$ to some *action*.

- An action is an algorithm which possibly returns some value and side-affects some data structures (the symbol table, for example).

Example

Example

In the sequel, *lex* is the lexeme matching a pattern.

$$
\begin{array}{lll}
ws & \longrightarrow & [\ \texttt{\textbackslash t\textbackslash n}]^{+} \\
 & & \mathcal{A}(ws) = \{\} \\
letter\_ & \longrightarrow & [\texttt{A} - \texttt{Z}\texttt{a} - \texttt{z}] \mid \_ \\
digit & \longrightarrow & [\texttt{0} - \texttt{9}] \\
id & \longrightarrow & letter\_(letter\_|digit)^{*} \\
 & & \mathcal{A}(id) = \{return(\langle \mathbf{id}, consultTable(lex)\rangle)\} \\
number & \longrightarrow & (digit)^{+}\ (.\ digit^{+})?\ (\texttt{E}\ [+-]?\ digit^{+})? \\
 & & \mathcal{A}(number) = \{return\langle \mathbf{num}, lex\rangle\}
\end{array}
$$

Fallback DFA with Actions

Definition

A fallback DFA with actions is a 6-tuple $\langle Q, \Sigma, \delta, q_0, F, A \rangle$, where

- Q, Σ, δ, q_0, and F are as usual; and
- A maps every $q \in Q$ into an action.

Anatomy

- A fallback DFA with actions consists of a finite control, an infinite (to the right) tape, a stack, and two heads: L and R.
- Both heads are read-only heads.
- Initially, both heads point at the left-most tape cell, where the input starts.
- R can move only to the right.
- L can move to the right and to the left as explained in the sequel.

Anatomy

- A fallback DFA with actions consists of a finite control, an infinite (to the right) tape, a stack, and two heads: L and R.
- Both heads are read-only heads.
- Initially, both heads point at the left-most tape cell, where the input starts.
- R can move only to the right.
- L can move to the right and to the left as explained in the sequel.

Anatomy

- A fallback DFA with actions consists of a finite control, an infinite (to the right) tape, a stack, and two heads: L and R.
- Both heads are read-only heads.
- Initially, both heads point at the left-most tape cell, where the input starts.
- R can move only to the right.
- L can move to the right and to the left as explained in the sequel.

Anatomy

- A fallback DFA with actions consists of a finite control, an infinite (to the right) tape, a stack, and two heads: L and R.
- Both heads are read-only heads.
- Initially, both heads point at the left-most tape cell, where the input starts.
- R can move only to the right.
- L can move to the right and to the left as explained in the sequel.

Anatomy

- A fallback DFA with actions consists of a finite control, an infinite (to the right) tape, a stack, and two heads: L and R.
- Both heads are read-only heads.
- Initially, both heads point at the left-most tape cell, where the input starts.
- R can move only to the right.
- L can move to the right and to the left as explained in the sequel.

Operation

- A fallback DFA with actions operates like the standard DFA, moving only L, and pushing every state it enters onto the stack with every transition.
- This continues until the DFA runs out of input.
- If it runs out of input in state $q_a \in F$, it executes $A(q_a)$ and halts.
- If it runs out of input in $q_r \notin F$, it
  1. continues to simultaneously pop the stack and move L one step to the left until the stack gets empty or some $q_a \in F$ is popped.
  2. In the first case, the DFA executes $A(q_r)$ and halts.
  3. In the second case it does the following.

Operation

- A fallback DFA with actions operates like the standard DFA, moving only L, and pushing every state it enters onto the stack with every transition.
- This continues until the DFA runs out of input.
- If it runs out of input in state $q_a \in F$, it executes $A(q_a)$ and halts.
- If it runs out of input in $q_r \notin F$, it
  1. continues to simultaneously pop the stack and move L one step to the left until the stack gets empty or some $q_a \in F$ is popped.
  2. In the first case, the DFA executes $A(q_r)$ and halts.
  3. In the second case it does the following.
     1. Executes $A(q_a)$ (with *lex* being the string extending from R to L).
     2. Moves L one step to the right.
     3. Moves R to where L is.
     4. Empties the stack.
     5. Enters q_0.

Operation

- A fallback DFA with actions operates like the standard DFA, moving only L, and pushing every state it enters onto the stack with every transition.
- This continues until the DFA runs out of input.
- If it runs out of input in state $q_a \in F$, it executes $A(q_a)$ and halts.
- If it runs out of input in $q_r \notin F$, it
  1. continues to simultaneously pop the stack and move L one step to the left until the stack gets empty or some $q_a \in F$ is popped.
  2. In the first case, the DFA executes $A(q_r)$ and halts.
  3. In the second case it does the following.

Operation

- A fallback DFA with actions operates like the standard DFA, moving only L, and pushing every state it enters onto the stack with every transition.
- This continues until the DFA runs out of input.
- If it runs out of input in state $q_a \in F$, it executes $A(q_a)$ and halts.
- If it runs out of input in $q_r \notin F$, it
  1. continues to simultaneously pop the stack and move L one step to the left until the stack gets empty or some $q_a \in F$ is popped.
  2. In the first case, the DFA executes $A(q_r)$ and halts.
  3. In the second case it does the following.
     1. Executes $A(q_a)$ (with *lex* being the string extending from R to L).
     2. Moves L one step to the right.
     3. Moves R to where L is.
     4. Empties the stack.
     5. Enters q_0.

Operation

- A fallback DFA with actions operates like the standard DFA, moving only L, and pushing every state it enters onto the stack with every transition.
- This continues until the DFA runs out of input.
- If it runs out of input in state $q_a \in F$, it executes $A(q_a)$ and halts.
- If it runs out of input in $q_r \notin F$, it
  1. continues to simultaneously pop the stack and move L one step to the left until the stack gets empty or some $q_a \in F$ is popped.
  2. In the first case, the DFA executes $A(q_r)$ and halts.
  3. In the second case it does the following.
     1. Executes $A(q_a)$ (with *lex* being the string extending from R to L).
     2. Moves L one step to the right.
     3. Moves R to where L is.
     4. Empties the stack.
     5. Enters q_0.

Operation

- A fallback DFA with actions operates like the standard DFA, moving only L, and pushing every state it enters onto the stack with every transition.
- This continues until the DFA runs out of input.
- If it runs out of input in state $q_a \in F$, it executes $A(q_a)$ and halts.
- If it runs out of input in $q_r \notin F$, it
  1. continues to simultaneously pop the stack and move L one step to the left until the stack gets empty or some $q_a \in F$ is popped.
  2. In the first case, the DFA executes $A(q_r)$ and halts.
  3. In the second case it does the following.
     1. Executes $A(q_a)$ (with *lex* being the string extending from R to L).
     2. Moves L one step to the right.
     3. Moves R to where L is.
     4. Empties the stack.
     5. Enters q_0.

Operation

- A fallback DFA with actions operates like the standard DFA, moving only L, and pushing every state it enters onto the stack with every transition.
- This continues until the DFA runs out of input.
- If it runs out of input in state $q_a \in F$, it executes $A(q_a)$ and halts.
- If it runs out of input in $q_r \notin F$, it
  1. continues to simultaneously pop the stack and move L one step to the left until the stack gets empty or some $q_a \in F$ is popped.
  2. In the first case, the DFA executes $A(q_r)$ and halts.
  3. In the second case it does the following.
     1. Executes $A(q_a)$ (with *lex* being the string extending from R to L).
     2. Moves L one step to the right.
     3. Moves R to where L is.
     4. Empties the stack.
     5. Enters q_0.

Operation

- A fallback DFA with actions operates like the standard DFA, moving only L, and pushing every state it enters onto the stack with every transition.
- This continues until the DFA runs out of input.
- If it runs out of input in state $q_a \in F$, it executes $A(q_a)$ and halts.
- If it runs out of input in $q_r \notin F$, it
  1. continues to simultaneously pop the stack and move L one step to the left until the stack gets empty or some $q_a \in F$ is popped.
  2. In the first case, the DFA executes $A(q_r)$ and halts.
  3. In the second case it does the following.
     1. Executes $A(q_a)$ (with *lex* being the string extending from R to L).
     2. Moves L one step to the right.
     3. Moves R to where L is.
     4. Empties the stack.
     5. Enters q_0.

Operation

- A fallback DFA with actions operates like the standard DFA, moving only L, and pushing every state it enters onto the stack with every transition.
- This continues until the DFA runs out of input.
- If it runs out of input in state $q_a \in F$, it executes $A(q_a)$ and halts.
- If it runs out of input in $q_r \notin F$, it
  1. continues to simultaneously pop the stack and move L one step to the left until the stack gets empty or some $q_a \in F$ is popped.
  2. In the first case, the DFA executes $A(q_r)$ and halts.
  3. In the second case it does the following.
     1. Executes $A(q_a)$ (with *lex* being the string extending from R to L).
     2. Moves L one step to the right.
     3. Moves R to where L is.
     4. Empties the stack.
     5. Enters q_0.

Operation

- A fallback DFA with actions operates like the standard DFA, moving only L, and pushing every state it enters onto the stack with every transition.
- This continues until the DFA runs out of input.
- If it runs out of input in state $q_a \in F$, it executes $A(q_a)$ and halts.
- If it runs out of input in $q_r \notin F$, it
  1. continues to simultaneously pop the stack and move L one step to the left until the stack gets empty or some $q_a \in F$ is popped.
  2. In the first case, the DFA executes $A(q_r)$ and halts.
  3. In the second case it does the following.
     1. Executes $A(q_a)$ (with *lex* being the string extending from R to L).
     2. Moves L one step to the right.
     3. Moves R to where L is.
     4. Empties the stack.
     5. Enters q_0.

Operation

- A fallback DFA with actions operates like the standard DFA, moving only L, and pushing every state it enters onto the stack with every transition.
- This continues until the DFA runs out of input.
- If it runs out of input in state $q_a \in F$, it executes $A(q_a)$ and halts.
- If it runs out of input in $q_r \notin F$, it
    1. continues to simultaneously pop the stack and move L one step to the left until the stack gets empty or some $q_a \in F$ is popped.
    2. In the first case, the DFA executes $A(q_r)$ and halts.
    3. In the second case it does the following.
        1. Executes $A(q_a)$ (with *lex* being the string extending from R to L).
        2. Moves L one step to the right.
        3. Moves R to where L is.
        4. Empties the stack.
        5. Enters q_0.

Operation

- A fallback DFA with actions operates like the standard DFA, moving only L, and pushing every state it enters onto the stack with every transition.
- This continues until the DFA runs out of input.
- If it runs out of input in state $q_a \in F$, it executes $A(q_a)$ and halts.
- If it runs out of input in $q_r \notin F$, it
  1. continues to simultaneously pop the stack and move L one step to the left until the stack gets empty or some $q_a \in F$ is popped.
  2. In the first case, the DFA executes $A(q_r)$ and halts.
  3. In the second case it does the following.
     1. Executes $A(q_a)$ (with *lex* being the string extending from R to L).
     2. Moves L one step to the right.
     3. Moves R to where L is.
     4. Empties the stack.
     5. Enters q_0.

How to Build a Lexical Analyzer in Five Steps (I)

1. Write an action-augmented regular definition where, for each of the n lexical categories of our language L, there is a pair $\langle D_i, R_i \rangle$, where D_i is the category name and R_i is its pattern, with $D_i \in \mathcal{C}$.
   - Actions should produce the appropriate tokens and update the symbol table as needed.

How to Build a Lexical Analyzer in Five Steps (II)

2. For each $c \in \mathcal{C}$ compile a corresponding regular expression R_c^Σ over Σ.
   - Each R_c^Σ is associated with the pair $\langle c, \mathcal{A}(c) \rangle$.

How to Build a Lexical Analyzer in Five Steps (III)

3. Construct the regular expression $R = (R^{\Sigma}_{c_1} | R^{\Sigma}_{c_2} | \cdots | R^{\Sigma}_{c_m})$, where $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$.
   - Note the absence of $+$.

How to Build a Lexical Analyzer in Five Steps (IV)

4. Construct an NFA N which is equivalent to R, provided that
    1. In constructing an NFA N_i equivalent to $R^{\Sigma}_{c_i}$, make sure that N_i has a unique accept state.
    2. The unique accept state of N_i has the label c_i.

How to Build a Lexical Analyzer in Five Steps (V)

5. Construct a fallback DFA with actions, M, which is equivalent to N, provided that

1. For every $q_a \in F$, $A(q_a) = \mathcal{A}(c_i)$, where
    1. $c_i \in q_a$ and
    2. if $c_j \in q_a$, then $i \leq j$.
2. For every $q_r \notin F$, $A(q_a)$ is a suitable "error action."

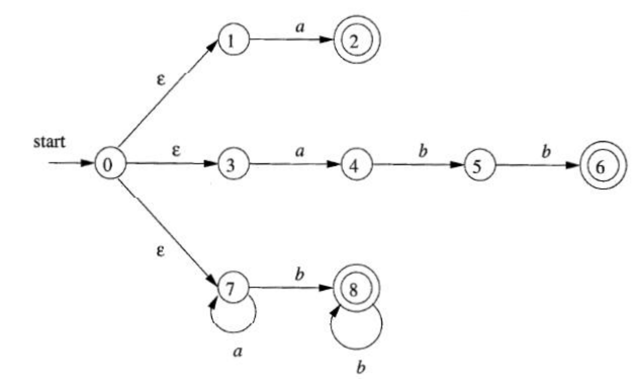
Example (I)

Example (Action-Augmented Regular Definition)

$$
\begin{array}{lll}
2 & \longrightarrow & \texttt{a} \\
 & & \mathcal{A}(2) = \{return(\langle \mathbf{2}, lex \rangle)\} \\
6 & \longrightarrow & \texttt{abb} \\
 & & \mathcal{A}(6) = \{return(\langle \mathbf{6}, lex \rangle)\} \\
8 & \longrightarrow & \texttt{a}^*\texttt{b}^+ \\
 & & \mathcal{A}(8) = \{return(\langle \mathbf{8}, lex \rangle)\}
\end{array}
$$

Example (II)

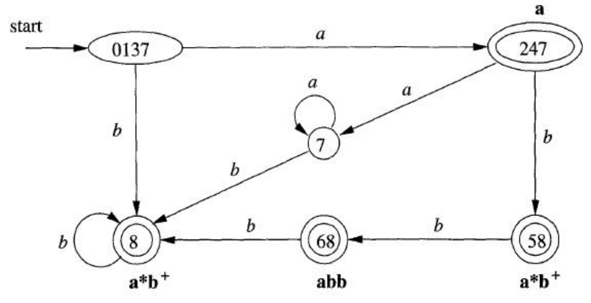
Example (Equivalent NFA)



©Aho et al. (2007)

Example (III)

Example (Fallback DFA with Actions)



©Aho et al. (2007)

What happens on input `abba`?