CS320 Principles of Programming Languages

Week 2: Syntax Specification, Grammars

Jingke Li

Portland State University

Fall 2017
Words and Sentences

Recall

- **Language** = \{ Sentences \}
- **Sentence** = Sequence of words
- **Grammar** = Rules for constructing sentences
Words and Sentences

Recall

- Language = \{ Sentences \}
- Sentence = Sequence of words
- Grammar = Rules for constructing sentences

Questions:

1. What information about words does a grammar need?
Words and Sentences

Recall

- Language = \{ Sentences \}
- Sentence = Sequence of words
- Grammar = Rules for constructing sentences

Questions:

1. What information about words does a grammar need?
2. Do we need rules for specifying words?
Words and Sentences

Let’s look at an example.

**Grammar:**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence</td>
<td>NounPhrase VerbPhrase</td>
</tr>
<tr>
<td>NounPhrase</td>
<td>Noun</td>
</tr>
<tr>
<td>VerbPhrase</td>
<td>Verb</td>
</tr>
</tbody>
</table>
Words and Sentences

Let’s look at an example.

**Grammar:**

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence</td>
<td>NounPhrase VerbPhrase</td>
</tr>
<tr>
<td>NounPhrase</td>
<td>Noun</td>
</tr>
<tr>
<td>VerbPhrase</td>
<td>Verb</td>
</tr>
</tbody>
</table>

**Sentences:**

(1) “Dogs run.”

“Birds fly.”

“Fish swim.”
Words and Sentences

Let’s look at an example.

**Grammar:**

\[
\text{Sentence} = \text{NounPhrase} \ \text{VerbPhrase} \ | \ \cdots \\
\text{NounPhrase} = \text{Noun} \ | \ \text{Article Noun} \ | \ \cdots \\
\text{VerbPhrase} = \text{Verb} \ | \ \text{Verb PrepPhrase} \ | \ \cdots 
\]

**Sentences:**

(1) “Dogs run.”  (2) “Fish run.”

“Birds fly.”  “Dogs fly.”

“Fish swim.”  “Birds swim.”
Words and Sentences

Let’s look at an example.

Grammar:

\[
\begin{align*}
\text{Sentence} & = \text{NounPhrase} \text{ VerbPhrase} \mid \cdots \\
\text{NounPhrase} & = \text{Noun} \mid \text{Article Noun} \mid \cdots \\
\text{VerbPhrase} & = \text{Verb} \mid \text{Verb PrepPhrase} \mid \cdots 
\end{align*}
\]

Sentences:

(1) “Dogs run.”

“Birds fly.”

“Fish swim.”

(2) “Fish run.”

“Dogs fly.”

“Birds swim.”

Observation:

- Both group of sentences are *syntactically* valid.
Words and Sentences

Let’s look at an example.

Grammar:

Sentence = NounPhrase VerbPhrase | · · ·
NounPhrase = Noun | Article Noun | · · ·
VerbPhrase = Verb | Verb PrePhrase | · · ·

Sentences:

(1) “Dogs run.”
   “Birds fly.”
   “Fish swim.”

(2) “Fish run.”
   “Dogs fly.”
   “Birds swim.”

Observation:

▶ Both group of sentences are syntactically valid.
▶ Grammars do not deal with individual words, they deal with categories of words.
Words and Sentences

▶ For English, Categories of words = Parts of Speech, i.e.
   Nouns, Pronouns, Verbs, Adjectives, Adverbs, Conjunctions, Prepositions, and Interjections
Words and Sentences

▶ For English, Categories of words = Parts of Speech, i.e.
   Nouns, Pronouns, Verbs, Adjectives, Adverbs, Conjunctions, Prepositions, and Interjections

▶ For PLs, Categories of words = Token Types, e.g.
   Keywords, Identifiers, Literals, Operators, etc.
Tokens

A program is built from tokens.

```c
/* A toy C program */
int main(void) {
    int a, b, s;
    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);
    s = a + 2 * b;
    printf("%d + 2*%d = %d\n", a, b, s);
}
```
A program is built from tokens.

/* A toy C program */
int main(void) {
    int a, b, s;
    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);
    s = a + 2 * b;
    printf("%d + 2*%d = %d\n", a, b, s);
}

Token = Smallest program entity that is meaningful to the language
Common Token Types

- **Keywords** — `int`, `void`, `printf`, `scanf`, ...
  - A finite set of language-defined names
Common Token Types

- **Keywords** — `int`, `void`, `printf`, `scanf`, ...
  - A finite set of language-defined names
  - Each keyword is typically given a token type of its own, since they are not interchangeable in a grammar
Common Token Types

- **Keywords** — `int`, `void`, `printf`, `scanf`, ...
  - A finite set of language-defined names
  - Each keyword is typically given a token type of its own, since they are not interchangeable in a grammar

- **Identifiers** — `main`, `a`, `b`, `s`, ...
  - A potentially infinite set of user-defined names
Common Token Types

- **Keywords** — `int, void, printf, scanf, ...`
  - A finite set of language-defined names
  - Each keyword is typically given a token type of its own, since they are not interchangeable in a grammar

- **Identifiers** — `main, a, b, s, ...`
  - A potentially infinite set of user-defined names
  - All IDs form a single token type
Common Token Types

▶ Keywords — int, void, printf, scanf, ...
  - A finite set of language-defined names
  - Each keyword is typically given a token type of its own, since they are not interchangeable in a grammar

▶ Identifiers — main, a, b, s, ...
  - A potentially infinite set of user-defined names
  - All IDs form a single token type

▶ Operators and Delimiters — (, ), {, }, ,, ;, =, +, *, ...
Common Token Types

- **Keywords** — `int, void, printf, scanf, ...`
  - A finite set of language-defined names
  - Each keyword is typically given a token type of its own, since they are not interchangeable in a grammar

- **Identifiers** — `main, a, b, s, ...`
  - A potentially infinite set of user-defined names
  - All IDs form a single token type

- **Operators and Delimiters** — `(, ), {, }, ,, ;, =, +, *, ...`
  - Each operator/delimiter is typically a separate token type
Common Token Types

- **Keywords** — `int`, `void`, `printf`, `scanf`, ...
  - A finite set of language-defined names
  - Each keyword is typically given a token type of its own, since they are not interchangeable in a grammar

- **Identifiers** — `main`, `a`, `b`, `s`, ...
  - A potentially infinite set of user-defined names
  - All IDs form a single token type

- **Operators and Delimiters** — `()`, `{}`, ` `, `,`, `;`, `==`, `+`, `*`, ...
  - Each operator/delimiter is typically a separate token type

- **Literals**
  - **Strings** — "Enter two integers: ", "%d %d", ...
  - **Integers** — 2, 365, ...
  - **Doubles** — 0.5, 3.14, ...
  ...
Common Token Types

- **Keywords** — `int`, `void`, `printf`, `scanf`, ...
  - A finite set of language-defined names
  - Each keyword is typically given a token type of its own, since they are not interchangeable in a grammar

- **Identifiers** — `main`, `a`, `b`, `s`, ...
  - A potentially infinite set of user-defined names
  - All IDs form a single token type

- **Operators and Delimiters** — `( )`, `{ }`, `,, ;`, `=`, `+`, `*`, ...
  - Each operator/delimiter is typically a separate token type

- **Literals**
  - **Strings** — "Enter two integers: ", "%d %d", ...
  - **Integers** — `2`, `365`, ...
  - **Doubles** — `0.5`, `3.14`, ...
  - Each literal group form a single token type
Lexemes

\[
\text{Lexeme} \quad = \quad \text{A particular, meaningful sequence of characters from a program}
\]
Lexemes

Lexeme = A particular, meaningful sequence of characters from a program

- Each token is associated with a lexeme.
Lexemes

Lexeme = A particular, meaningful sequence of characters from a program

- Each token is associated with a lexeme.
- A token type may correspond to a single lexeme —
  - Such as the cases of individual keywords, operators, and delimiters
  - In these cases, the token type is often named the same as the lexeme
Lexemes

A particular, meaningful sequence of characters from a program

- Each token is associated with a lexeme.
- A token type may correspond to a single lexeme —
  - Such as the cases of individual keywords, operators, and delimiters
  - In these cases, the token type is often named the same as the lexeme
- Or it may correspond to a collection of lexemes —
  - Such as the cases of IDs and literals
  - In these cases, the collection of lexemes typically share a common characteristics, e.g. integer literals are all “sequence of digits”
Non-Token Lexemes

- **Whitespace** — typically have no significance in the language other than to separate tokens:
  - the space, tab, newline character, ...
Non-Token Lexemes

- **Whitespace** — typically have no significance in the language other than to separate tokens:
  - the space, tab, newline character, ...

- **Comments** — have no effect on program’s semantics.
  - various forms depending on language
Non-Token Lexemes

- **Whitespace** — typically have no significance in the language other than to separate tokens:
  - the space, tab, newline character, ...

- **Comments** — have no effect on program’s semantics.
  - various forms depending on language

- **Illegal Characters** — characters that do no belong to the language’s alphabet.
  - # ~ $ ~ @
Non-Token Lexemes

- **Whitespace** — typically have no significance in the language other than to separate tokens:
  - the space, tab, newline character, ...

- **Comments** — have no effect on program's semantics.
  - various forms depending on language

- **Illegal Characters** — characters that do no belong to the language’s alphabet.
  - # ~ $ ~ @

- **Ill-Formed Tokens** — Not all legal characters form valid tokens.
  - 0789, 0xAG, "abc<EOF>
Non-Token Lexemes

- **Whitespace** — typically have no significance in the language other than to separate tokens:
  - the space, tab, newline character, ...

- **Comments** — have no effect on program's semantics.
  - various forms depending on language

- **Illegal Characters** — characters that do no belong to the language’s alphabet.
  - # ^ $ ~ @

- **Ill-Formed Tokens** — Not all legal characters form valid tokens.
  - 0789, 0xAG, "abc<EOF>

Whitespace and comments are filtered out during lexical analysis; while illegal chars and ill-formed tokens are *lexical errors*, which should caught and reported.
Common Comment Types

▶ Single line:

```
// C, C++, Java
-- Haskell
; Lisp, Scheme
C Fortran
# csh, bash, make
```
Common Comment Types

▶ Single line:

// C, C++, Java
-- Haskell
; Lisp, Scheme
C Fortran
# csh, bash, make

▶ Non-nesting block:

/* C, C++, Java */
Common Comment Types

▶ Single line:

// C, C++, Java
-- Haskell
; Lisp, Scheme
C Fortran
# csh, bash, make

▶ Non-nesting block:

/* C, C++, Java */

▶ Nesting block:

(* Pascal, ML (* allow nesting *) *)
{- Haskell {- as well -} -}
Token Representation

A token is typically represented as an object with attributes:
Token Representation

A token is typically represented as an object with *attributes*:

- **Type** — the key info used by a grammar
Token Representation

A token is typically represented as an object with attributes:

- **Type** — the key info used by a grammar
- **Lexeme** — to distinguish individual member of a group
Token Representation

A token is typically represented as an object with *attributes*:

- **Type** — the key info used by a grammar
- **Lexeme** — to distinguish individual member of a group
- **Value** — for numerical tokens
  - It is common for a compiler to convert a numerical token’s lexeme to a value, and store it in the token object (in addition to or instead of the lexeme).
Token Representation

A token is typically represented as an object with attributes:

- **Type** — the key info used by a grammar
- **Lexeme** — to distinguish individual member of a group
- **Value** — for numerical tokens
  - It is common for a compiler to convert a numerical token’s lexeme to a value, and store it in the token object (in addition to or instead of the lexeme).
- **Position** — line and column numbers (for diagnostic use)
Rules for Specifying Words?
Rules for Specifying Words?

- For English, the vocabulary ($\approx 172,000$ words) is fixed
  - There is no need to create new words
  - Given a word, it is not hard to find the part of speech(s) it belongs to
    — Just look up a dictionary
Rules for Specifying Words?

- For English, the vocabulary (≈ 172,000 words) is fixed
  - There is no need to create new words
  - Given a word, it is not hard to find the part of speech(s) it belongs to
    — Just look up a dictionary

- For PLs, the vocabulary is *not* fixed
Rules for Specifying Words?

- For English, the vocabulary ($\approx 172,000$ words) is fixed
  - There is no need to create new words
  - Given a word, it is not hard to find the part of speech(s) it belongs to
    — Just look up a dictionary
- For PLs, the vocabulary is *not* fixed
  - Programmer needs to know how to create new tokens
Rules for Specifying Words?

- For English, the vocabulary (≈ 172,000 words) is fixed
  - There is no need to create new words
  - Given a word, it is not hard to find the part of speech(s) it belongs to
    — Just look up a dictionary

- For PLs, the vocabulary is *not* fixed
  - Programmer needs to know how to create new tokens
  - Compiler needs to know how to recognize tokens (and non-tokens) without a dictionary
Rules for Specifying Words?

▶ For English, the vocabulary (≈ 172,000 words) is fixed
  - There is no need to create new words
  - Given a word, it is not hard to find the part of speech(s) it belongs to
    — Just look up a dictionary

▶ For PLs, the vocabulary is *not* fixed
  - Programmer needs to know how to create new tokens
  - Compiler needs to know how to recognize tokens (and non-tokens)
    without a dictionary — the information must come from the lexemes themselves
Rules for Specifying Words?

- For English, the vocabulary ($\approx 172,000$ words) is fixed
  - There is no need to create new words
  - Given a word, it is not hard to find the part of speech(s) it belongs to
    - Just look up a dictionary

- For PLs, the vocabulary is *not* fixed
  - Programmer needs to know how to create new tokens
  - Compiler needs to know how to recognize tokens (and non-tokens) without a dictionary — the information must come from the lexemes themselves

We need a mechanism to *precisely* define tokens.
Patterns

Pattern = A description of lexical features for a set of character sequences
Patterns

Pattern = A description of lexical features for a set of character sequences

Examples:
- Sequence of digits — 1234, 503, 00001
- Sequence within quotes — "1234", "abc", "@+-*"
- Sequence starts with letter 'a' — a1234, abc, a@+-*
Patterns

Pattern = A description of lexical features for a set of character sequences

Examples:
- Sequence of digits — 1234, 503, 00001
- Sequence within quotes — "1234", "abc", "@+-*"
- Sequence starts with letter ’a’ — a1234, abc, a@+-*

▷ For a PL, lexemes belonging to the same token type typically share a common pattern.
Patterns

\[
\text{Pattern} = \text{A description of lexical features for a set of character sequences}
\]

**Examples:**

- Sequence of digits — 1234, 503, 00001
- Sequence within quotes — "1234", "abc", "@+-*"
- Sequence starts with letter 'a' — a1234, abc, a@+-*

For a PL, lexemes belonging to the same token type typically share a common pattern.

The preferred tool for precisely specifying patterns is regular expression.
Review: Regular Expressions

RE defines strings over a finite alphabet with three operations:

- \( | \) — alternate
- \( \cdot \) — concatenate (omitable)
- \( \ast \) — repeat 0 or more times
Review: Regular Expressions

RE defines strings over a finite alphabet with three operations:
- | — alternate
- · — concatenate (omitable)
- * — repeat 0 or more times

Additional operations are available:
- † — repeat 1 or more times
- ∼ — complement w.r.t. an alphabet (or an expression*)
- − — difference

... (and more) ...
Review: Regular Expressions

RE defines strings over a finite alphabet with three operations:

- | — alternate
- • — concatenate (omitable)
- * — repeat 0 or more times

Additional operations are available:

- + — repeat 1 or more times
- ~ — complement w.r.t. an alphabet (or an expression*)
- − — difference

... (and more) ...

Examples: abc, ab|c*, (a|b)(c|d), ab[^ab]*, a*b*c*−aaa

* Complement of expression is typically not available in RE tools due to its complexity.
Tokens with single lexeme, such as keywords, operators, and delimiters, are easy:

<table>
<thead>
<tr>
<th>Token</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>&quot;int&quot;</td>
</tr>
<tr>
<td>Void</td>
<td>&quot;void&quot;</td>
</tr>
<tr>
<td>PLUS</td>
<td>&quot;+&quot;</td>
</tr>
<tr>
<td>MINUS</td>
<td>&quot;-&quot;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Specifying Token Patterns with RE

- Tokens with single lexeme, such as keywords, operators, and delimiters, are easy:

<table>
<thead>
<tr>
<th>Token</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>&quot;int&quot;</td>
</tr>
<tr>
<td>Void</td>
<td>&quot;void&quot;</td>
</tr>
<tr>
<td>PLUS</td>
<td>&quot;+&quot;</td>
</tr>
<tr>
<td>MINUS</td>
<td>&quot;-&quot;</td>
</tr>
</tbody>
</table>

- The ID pattern has one small issue: an ID cannot have the same spelling as a keyword. We can use the difference operation to solve this problem:

```
Digit = [0-9]
Letter = [A-Z|a-z]
KWD = Int | Void | ...
ID = Letter (Letter | Digit | ")* - KWD
```

* Compiler typically uses a different approach.
Specifying Token Patterns with RE

- String literal is straightforward:

\[
\text{StrLit} = "\\" (~["\\","\\n"])+ "\\"
\]
Specifying Token Patterns with RE

- **String literal is straightforward:**

  
  ```
  StrLit = "\"" (~["\"", "\n"])+ "\"
  ```

- **Numerical literals can be a little tricky:**
  - For integers, there can be multiple forms: decimal, octal, and hex.
  - For floating-point numbers, both the integral part and the fractional part can be empty, *e.g.* 123. and .123, yet they can’t be empty at the same time.
Specifying Token Patterns with RE

- **String literal is straightforward:**
  
  \[
  \text{StrLit} = \"\" (~["\"","\n"])+ \"\"
  \]

- **Numerical literals can be a little tricky:**
  - For integers, there can be multiple forms: decimal, octal, and hex.
  - For floating-point numbers, both the integral part and the fractional part can be empty, *e.g.* 123. and .123, yet they can’t be empty at the same time.

  \[
  \begin{align*}
  \text{OctLit} & = 0([0-7])+ \\
  \text{HexLit} & = \"0x\" ([0-9a-f])+ \\
  \text{DecLit} & = \text{Digit}+ - \text{OctLit} \\
  \text{IntLit} & = \text{DecLit} \mid \text{OctLit} \mid \text{HexLit} \\
  \text{RealLit} & = \text{Digit}+ \text{.} \text{Digit}* \mid \text{Digit}* \text{.} \text{Digit}+
  \end{align*}
  \]
Is RE Perfect for Token Specification?

► Consider C’s non-nesting block comments:

“A block comment begins with the characters /* and ends with the first subsequent occurrence of the characters */.”
Consider C’s non-nesting block comments:

“A block comment begins with the characters /* and ends with the first subsequent occurrence of the characters */.”

The following is a direct encoding of the definition in RE:

```
BlockComment = "/\*" \( (~("/\*" | <EOF>)) \)* "*/" >
```
Is RE Perfect for Token Specification?

- Consider C’s non-nesting block comments:
  “A block comment begins with the characters */ and ends with the first subsequent occurrence of the characters */.”

The following is a direct encoding of the definition in RE:

```
BlockComment = "/\*" \((\"*/\"|<EOF>)\)* \\"*/\"">
```

It uses the extended complement operator. While theoretically correct, most RE processing systems don’t allow it. (Why?)
Is RE Perfect for Token Specification?

Consider C’s non-nesting block comments:

“A block comment begins with the characters /* and ends with the first subsequent occurrence of the characters */.”

The following is a direct encoding of the definition in RE:

```
BlockComment = "/\"" (\"\/*\"|<EOF>)\"/\"\n```

It uses the extended complement operator. While theoretically correct, most RE processing systems don’t allow it. (*Why?*)

It is harder to define BlockComment without using this operator.
Is RE Perfect for Token Specification?

Consider C’s non-nesting block comments:
“A block comment begins with the characters /* and ends with the first subsequent occurrence of the characters */.”

The following is a direct encoding of the definition in RE:

```
BlockComment = "/*" (~("*/"|<EOF>))* "/"">
```

It uses the extended complement operator. While theoretically correct, most RE processing systems don’t allow it. (Why?)

It is harder to define BlockComment without using this operator.

Now consider the pattern of ML’s nesting block comments:
“Comments in Standard ML begin with (* and end with *). Comments can be nested which means that all (* tags must end with a *) tag.”
Is RE Perfect for Token Specification?

Consider C’s non-nesting block comments:
“A block comment begins with the characters /* and ends with the first subsequent occurrence of the characters */.”

The following is a direct encoding of the definition in RE:

```
BlockComment = "/*" (~("/*/"|<EOF>))* "*/">
```

It uses the extended complement operator. While theoretically correct, most RE processing systems don’t allow it. (*Why?*)

It is harder to define BlockComment without using this operator.

Now consider the pattern of ML’s nesting block comments:
“Comments in Standard ML begin with (* and end with *). Comments can be nested which means that all (* tags must end with a *) tag.”

*Question:* Can you write a regular expression for it?
Review: Finite Automata

For every regular set \( L \), there exists a finite automaton that accepts \( L \).

*Example: \( aa^*|bb^* \)

- Circles represent *states*; double-circles represent *final* states.
- Labeled edges represent *transitions*. 
Review: Finite Automata

For every regular set $L$, there exists a finite automaton that accepts $L$.

**Example:** $aa^* | bb^*$

![Diagram of a finite automaton with states labeled with circles and transitions labeled with edges](image)

- Circles represent *states*; double-circles represent *final* states.
- Labeled edges represent *transitions*.

A finite automaton accepts string $x$ if there is a path from the start state to a final state labeled by characters of $x$. A finite automaton accepts language $L$ if it accepts exactly the strings of $L$. 
NFA vs. DFA

RE: $aa^* | bb^*$
NFA vs. DFA

RE: $aa^* | bb^*$

- **NFA:**
  - Transitions may be labeled by $\epsilon$
  - Multiple transitions from the same state may be labeled by the same symbol
NFA vs. DFA

RE: $aa^*|bb^*$

- **NFA:**
  - Transitions may be labeled by $\epsilon$
  - Multiple transitions from the same state may be labeled by the same symbol

- **DFA:**
  - No $\epsilon$-transitions
  - Each state has at most one transition labeled by the same symbol
Transition Graph vs. Table Form

RE: \((a|b)^*abb\)

- **NFA:**

- **DFA:**

<table>
<thead>
<tr>
<th>State</th>
<th>(\epsilon)</th>
<th>(a)</th>
<th>(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>(a)</th>
<th>(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Using FA to Recognize Tokens

Assume tokens are defined by RE.

Steps:
1. Construct an NFA based on the token REs.
2. Convert the NFA to a DFA.
3. (Optional) Minimize the DFA’s states.
4. Implement the DFA as a lexer program.
Using FA to Recognize Tokens

Assume tokens are defined by RE.

Steps:
1. Construct an NFA based on the token REs.
2. Convert the NFA to a DFA.
3. (Optional) Minimize the DFA’s states.
4. Implement the DFA as a lexer program.

We’ll use \((a|b)^*abb\) as an example.
Step 1. RE $\Rightarrow$ NFA

$a|b$: 

(a|b)*: 

$abb$: 

$(a|b)^*abb$: 

\[
\begin{align*}
(a|b)^* & : \\
(a|b) & : \\
\end{align*}
\]
Step 2. NFA $\Rightarrow$ DFA

NFA:

Find the $\epsilon$-closure for the start state 0:

Each state in the DFA corresponds to a set of states in the NFA.
Step 2. NFA ⇒ DFA (cont.)

NFA:

Find the target states corresponding to symbols $a$ and $b$:

For each symbol, the DFA’s transition simulates all possible transitions in the NFA.
Step 2. NFA ⇒ DFA (cont.)

NFA:

- Repeat the process for the new states:
Step 2. NFA ⇒ DFA (cont.)

NFA:

Finally:
Step 2. NFA $\Rightarrow$ DFA (cont.)

- The final DFA:

```
<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
```
A DFA implementing token REs can be easily turned into a program that answers “yes” or “no” on whether a given input sequence represents a token:

- Traverse the DFA with the input and see if it reaches a final state.
Step 4. DFA $\Rightarrow$ Lexer Program

A DFA implementing token REs can be easily turned into a program that answers “yes” or “no” on whether a given input sequence represents a token:

- Traverse the DFA with the input and see if it reaches a final state.

**Examples:**

- a b b — “yes”, since it reaches a final state.
Step 4. DFA ⇒ Lexer Program

A DFA implementing token REs can be easily turned into a program that answers “yes” or “no” on whether a given input sequence represents a token:

- Traverse the DFA with the input and see if it reaches a final state.

Examples:

- a b b — “yes”, since it reaches a final state.
- b a a — “no”, since it does not reach a final state.
Step 4. DFA ⇒ Lexer Program (cont.)

However, if the input sequence may represent multiple tokens, there is a problem — when should the program report “yes”?

- Should it report as soon as the transitions reach a final state?
- Or should it keep on reading more input before reporting?
Step 4. DFA $\Rightarrow$ Lexer Program (cont.)

However, if the input sequence may represent multiple tokens, there is a problem — when should the program report “yes”?

- Should it report as soon as the transitions reach a final state?
- Or should it keep on reading more input before reporting?

**Examples:**

$a \ b \ b \ a \ b \ b$ — one or two tokens?
Step 4. DFA $\Rightarrow$ Lexer Program (cont.)

However, if the input sequence may represent multiple tokens, there is a problem — when should the program report “yes”?

- Should it report as soon as the transitions reach a final state?
- Or should it keep on reading more input before reporting?

Examples:

a b b a b b — one or two tokens?

a b b a b — one token plus some garbage or a single piece of garbage?
Longest Match/Maximal Munch Rule

A widely used convention for token recognition.

- If multiple lexemes can be formed from the same input point, the longest one will be chosen.

```
int 1 2 3  — one ID token
```
Longest Match/Maximal Munch Rule

A widely used convention for token recognition.

- If multiple lexemes can be formed from the same input point, the longest one will be chosen.

  \texttt{int123} — one ID token

- A white space (including tab, newline, and comment, etc.) automatically terminates a lexeme.

  \texttt{int12 \[3] } — one ID token followed by one INTLIT token
Longest Match/Maximal Munch Rule

A widely used convention for token recognition.

► If multiple lexemes can be formed from the same input point, the longest one will be chosen.

\[\text{int123} \quad \text{— one ID token}\]

► A white space (including tab, newline, and comment, etc.) automatically terminates a lexeme.

\[\text{int12 }\text{ [] }3 \quad \text{— one ID token followed by one INTLIT token}\]

► (Priority/First-Match Rule) If the longest lexeme can be matched with multiple tokens, the token whose definition appears first in the token specification file has the priority.

\[\text{begin} \quad \text{— keyword or ID?}\]
Longest Match/Maximal Munch Rule

A widely used convention for token recognition.

- If multiple lexemes can be formed from the same input point, the longest one will be chosen.
  
  ```
  int123  — one ID token
  ```

- A white space (including tab, newline, and comment, etc.) automatically terminates a lexeme.
  
  ```
  int12 3  — one ID token followed by one INTLIT token
  ```

- (Priority/First-Match Rule) If the longest lexeme can be matched with multiple tokens, the token whose definition appears first in the token specification file has the priority.
  
  ```
  begin  — keyword or ID?
  ```

(Corollary) In a token definition file, always define keywords before ID.
Longest Match/Maximal Munch Rule — Exercises

Show token sequence for each of the following Java input:

- x +++ y
- x +++ + y
- x +++ + + y
- x +++ + + + y
- x ++ + + + + y
- x ++ + + + + y
- x + + + ++ + y
- x + + + + + y
- x + + + + + + y
Show token sequence for each of the following Java input:

- x + + + y
  - ID(x) "++" "+" ID(y)
- x + + + + y
- x + + + + + y
- x □ + + + + + y
- x + + + □ + + y
- x + □ + □ + □ + + y
Longest Match/Maximal Munch Rule — Exercises

Show token sequence for each of the following Java input:

- ▶ x +++ y
  
  ID(x) "++" "+" ID(y)

- ▶ x ++++ y
  
  ID(x) "++" "++" ID(y)

- ▶ x ++++ + y

- ▶ x + + + + + y

- ▶ x + + + + + + y

- ▶ x + + + □ + + y

- ▶ x + + + + □ + + y

- ▶ x + □ + □ + □ + + y
Show token sequence for each of the following Java input:

- `x +++ y`  
  ID(x) "++" "++" ID(y)
- `x ++++ y`  
  ID(x) "++" "++" ID(y)
- `x ++++ y`  
  ID(x) "++" "++" ID(y)
- `x +++++ y`  
  ID(x) "++" "++" "++" ID(y)
- `x ++++ y`  
  ID(x) "++" "++" ID(y)
- `x +++++ y`  
  ID(x) "++" "++" "++" ID(y)
- `x +++++ y`  
  ID(x) "++" "++" "++" ID(y)
- `x +++++ y`  
  ID(x) "++" "++" "++" ID(y)
Show token sequence for each of the following Java input:

- $x + + + y$
  
  - ID(x) "++" "+" ID(y)

- $x + + + + y$
  
  - ID(x) "++" +++ ID(y)

- $x + + + + + y$
  
  - ID(x) "++" +++ +++ ID(y)

- $x + + + ++ y$
  
  - ID(x) "++" +++ ++ ID(y)

- $x + + ++ + + y$
  
  - ID(x) "++" +++ ++ ++ ID(y)
Longest Match/Maximal Munch Rule — Exercises

Show token sequence for each of the following Java input:

- \( x + + + y \)
  \[ \text{ID}(x) \ "++" \ "+" \ \text{ID}(y) \]

- \( x + + + + y \)
  \[ \text{ID}(x) \ "++" \ "++" \ \text{ID}(y) \]

- \( x + + + + + y \)
  \[ \text{ID}(x) \ "++" \ "++" \ "+" \ \text{ID}(y) \]

- \( x + + + + + + y \)
  \[ \text{ID}(x) \ "++" \ "++" \ "++" \ "+" \ \text{ID}(y) \]

- \( x + + + + + + + y \)
  \[ \text{ID}(x) \ "++" \ "++" \ "++" \ "+" \ \text{ID}(y) \]

- \( x + \Box + + + + + y \)
  \[ \text{ID}(x) \ "++" \ "++" \ "++" \ "+" \ \text{ID}(y) \]

- \( x + + + \Box + + y \)
  \[ \text{ID}(x) \ "++" \ "+" \ "++" \ \text{ID}(y) \]

- \( x + \Box + \Box + \Box + + y \)
  \[ \text{ID}(x) \ "++" \ "+" \ "++" \ "++" \ \text{ID}(y) \]
Show token sequence for each of the following Java input:

- x ++ y
  - ID(x) "++" ID(y)
- x +++ y
  - ID(x) "++" "++" ID(y)
- x ++++ y
  - ID(x) "++" "++" "++" ID(y)
- x +++++ y
  - ID(x) "++" "++" "++" ID(y)
- x ++ ++ ++ y
  - ID(x) "++" "++" "++" "++" ID(y)
- x +++ + ++ y
  - ID(x) "++" "++" "++" "++" ID(y)
- x + + + + + y
  - ID(x) "++" "++" "++" "++" "++" ID(y)
Dealing with Difficult Patterns

An alternative way of finding a regular expression for a pattern is to find a finite automata first, then convert the FA to an RE.
Dealing with Difficult Patterns

An alternative way of finding a regular expression for a pattern is to find a finite automata first, then convert the FA to an RE.

*Example:* Find a regular expression for the set of sequences of digits whose values are multiples of 3. *E.g.* 3, 060, 12345 are all in the set, while 11, 22, 2345 are not.
Dealing with Difficult Patterns

An alternative way of finding a regular expression for a pattern is to find a finite automata first, then convert the FA to an RE.

**Example:** Find a regular expression for the set of sequences of digits whose values are multiples of 3. *E.g.* 3, 060, 12345 are all in the set, while 11, 22, 2345 are not.

**Analysis:**
- On the surface, the set seems to be defined through a semantic property, *i.e.*, *value of sequence*. 
Dealing with Difficult Patterns

An alternative way of finding a regular expression for a pattern is to find a finite automata first, then convert the FA to an RE.

*Example:* Find a regular expression for the set of sequences of digits whose values are multiples of 3. *E.g.* 3, 060, 12345 are all in the set, while 11, 22, 2345 are not.

*Analysis:*

- On the surface, the set seems to be defined through a semantic property, *i.e.*, *value of sequence*.
- However, mathematical knowledge tells us that this particular semantic property can be mapped to an equivalent *syntactical* property:

<table>
<thead>
<tr>
<th>A sequence of digits whose value is a multiple of 3</th>
<th>The sum of the sequence’s digits is a multiple of 3</th>
</tr>
</thead>
</table>
Dealing with Difficult Patterns (cont.)

- Even with the new insight, it is still not easy to come up with a regular expression. (*It’s a good exercise to try.*)
Dealing with Difficult Patterns (cont.)

▶ Even with the new insight, it is still not easy to come up with a regular expression. (*It’s a good exercise to try.*)

▶ Now, let’s think from a finite automaton’s point of view. With respect to the property of sequence’s value being a multiples of 3, an arbitrary sequence of digits can only be in one of three states:

   - **S1.** It’s value is a multiple of 3 (so is the sum of its digits)
   - **S2.** It’s value is a multiple of 3 plus 1
   - **S3.** It’s value is a multiple of 3 plus 2
Dealing with Difficult Patterns (cont.)

With this info, we can define the following DFA for recognizing the set:
Dealing with Difficult Patterns (cont.)

With this info, we can define the following DFA for recognizing the set:

If the goal is to implement a lexer, we can go directly from here.
Dealing with Difficult Patterns (cont.)

With this info, we can define the following DFA for recognizing the set:

- If the goal is to implement a lexer, we can go directly from here.
- If the goal is to find a RE, we can follow an conversion algorithm.
Converting a DFA to a RE

Method: The state removal method.
Converting a DFA to a RE

**Method:** The state removal method.

**Step 1. (Preparation)** If the start state has an incoming edge from another state, create a new start state with an $\epsilon$ transition connecting to the original start state. If there are multiple final states, or a final state with an out-going edge, create a new final state with $\epsilon$ transitions connecting from all the original final states.
Converting a DFA to a RE

**Method:** The state removal method.

**Step 1. (Preparation)** If the start state has an incoming edge from another state, create a *new* start state with an $\epsilon$ transition connecting to the original start state. If there are multiple final states, or a final state with an out-going edge, create a *new* final state with $\epsilon$ transitions connecting from all the original final states.

**Step 2. (Elimination)** Pick a state. For every pair of incoming and out-going edges (excluding self-loop), consolidate the labels on the path to a single label, and create a direct edge with that label. Eliminate the state.
Converting a DFA to a RE

Method: The state removal method.

Step 1. (Preparation) If the start state has an incoming edge from another state, create a new start state with an $\epsilon$ transition connecting to the original start state. If there are multiple final states, or a final state with an out-going edge, create a new final state with $\epsilon$ transitions connecting from all the original final states.

Step 2. (Elimination) Pick a state. For every pair of incoming and out-going edges (excluding self-loop), consolidate the labels on the path to a single label, and create a direct edge with that label. Eliminate the state.

Repeat this step until done.
Converting a DFA to a RE

1. Preparation.

$L_0 = 0, 3, 6, 9$
$L_1 = 1, 4, 7$
$L_2 = 2, 5, 8$
Converting a DFA to a RE

1. Preparation.

$L_0 = 0, 3, 6, 9$
$L_1 = 1, 4, 7$
$L_2 = 2, 5, 8$
Converting a DFA to a RE

2. Eliminating State 2.
Converting a DFA to a RE

2. Eliminating State 2.

![DFA Diagram]

\[ L_0 + L_1 L_0^* L_2 \]

\[ L_0 + L_2 L_0^* L_1 \]
Converting a DFA to a RE

3. Eliminating State 3.
Converting a DFA to a RE

3. Eliminating State 3.

\[ \begin{align*}
0 & \rightarrow L_0 & & \rightarrow L_0 + L_1 L_0 * L_2 \\
1 & \rightarrow L_0 + L_1 L_0 * L_2 & & \rightarrow L_2 + L_1 L_0 * L_1 \\
3 & \rightarrow L_0 + L_2 L_0 * L_1 & & \rightarrow L_1 + L_2 L_0 * L_2
\end{align*} \]
Converting a DFA to a RE

4. Eliminating State 1.

\[
L_0 + L_1L_0^*L_2 + (L_2 + L_1L_0^*L_1)(L_0 + L_2L_0^*L_1)^*(L_1 + L_2L_0^*L_2)
\]
Converting a DFA to a RE

4. Eliminating State 1.

\[
L_0 + L_1 L_0 * L_2 + (L_2 + L_1 L_0 * L_1) (L_0 + L_2 L_0 * L_1) * (L_1 + L_2 L_0 * L_2)
\]

\[
L_0 = 0, 3, 6, 9
\]
\[
L_1 = 1, 4, 7
\]
\[
L_2 = 2, 5, 8
\]
Context-Free Grammars (BNF Notation)

BNF = Backus-Naur Form

Common convention:
- Nonterminals begin with upper-case letters.
- Literal terminals are quoted.
- Other terminals begin with lower-case letters.

Program → “begin” StmtList “end”
StmtList → Statement StmtList
StmtList → Statement
Statement → Assignment
Statement → ReadStmt
Statement → WriteStmt
Assignment → id “:=” Expression “;”
ReadStmt → “read” “(” IdList “)” “;”
WriteStmt → “write” “(” ExprList “)” “;”
IdList → id “,” IdList
IdList → id
ExprList → Expression “,” ExprList
ExprList → Expression
Expression → Primary “+” Expression
Expression → Primary “−” Expression
Expression → Primary
Primary → “(” Expression “)”
Primary → id
Primary → integer
Context-Free Grammars (EBNF Notation)

EBNF = Extended BNF

Common convention:
- `|` for selection
- `(,)` for grouping
- `[ ]` for an optional construct
- `{}` for zero or more repetitions

Program → “begin” StmtList “end”
StmtList → Statement {Statement}
Statement → Assignment
  | ReadStmt
  | WriteStmt
Assignment → id “:=” Expression “;”
ReadStmt → “read” “(” IdList “)” “;”
WriteStmt → “write” “(” ExprList “)” “;”
IdList → id { “,” id }
ExprList → Expression { “,” Expression }
Expression → Primary { (“+” | “−”) Primary }
Primary → “(” Expression “)”
  | id
  | integer
Equivalence Between EBNF and BNF

Extended BNF does not have extra power over BNF. In fact, any grammar in EBNF can be transformed into one in proper BNF:

\[
A \rightarrow \alpha [\beta] \gamma \quad \Rightarrow \quad A \rightarrow \alpha B \gamma \\
B \rightarrow \beta \\
B \rightarrow \epsilon
\]

\[
A \rightarrow \alpha \{\beta\} \gamma \quad \Rightarrow \quad A \rightarrow \alpha B \gamma \\
B \rightarrow \beta B \\
B \rightarrow \epsilon
\]
RE vs. CFG

Every set that can be described by a regular expression can also be described by a context-free grammar.

**Example:** The RE \((a | b)^* abb\) can be described by the CFG

\[
\begin{align*}
A_0 & \rightarrow aA_0 \mid bA_0 \mid aA_1 \\
A_1 & \rightarrow bA_2 \\
A_2 & \rightarrow bA_3 \\
A_3 & \rightarrow \epsilon
\end{align*}
\]

Regular grammars are just CFGs with their productions limited to the following two forms,

\[A \rightarrow aB \text{ and } C \rightarrow c\]
Derivations

A *derivation* step is to replace the lhs of a production by its rhs.
Derivations

A *derivation* step is to replace the lhs of a production by its rhs.

**Example:**

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T \ast P$
4. $T \rightarrow P$
5. $P \rightarrow \text{id}$
Derivations

A *derivation* step is to replace the lhs of a production by its rhs.

**Example:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$E \rightarrow E + T$</td>
</tr>
<tr>
<td>2.</td>
<td>$E \rightarrow T$</td>
</tr>
<tr>
<td>3.</td>
<td>$T \rightarrow T \ast P$</td>
</tr>
<tr>
<td>4.</td>
<td>$T \rightarrow P$</td>
</tr>
<tr>
<td>5.</td>
<td>$P \rightarrow id$</td>
</tr>
</tbody>
</table>

$E \dfrac{1}{\Rightarrow} E + T$
Derivations

A *derivation* step is to replace the lhs of a production by its rhs.

**Example:**

<table>
<thead>
<tr>
<th>Step</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$E \rightarrow E + T$</td>
</tr>
<tr>
<td>2.</td>
<td>$E \rightarrow T$</td>
</tr>
<tr>
<td>3.</td>
<td>$T \rightarrow T * P$</td>
</tr>
<tr>
<td>4.</td>
<td>$T \rightarrow P$</td>
</tr>
<tr>
<td>5.</td>
<td>$P \rightarrow \text{id}$</td>
</tr>
</tbody>
</table>

$E \xrightarrow{1} E + T \xrightarrow{2} T + T$
Derivations

A *derivation* step is to replace the lhs of a production by its rhs.

**Example:**

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * P$
4. $T \rightarrow P$
5. $P \rightarrow \text{id}$

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow T * P + T$
Derivations

A *derivation* step is to replace the lhs of a production by its rhs.

**Example:**

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \ast P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow \text{id} \)

\[
\begin{align*}
E & \Rightarrow E + T & \Rightarrow T + T \\
& \Rightarrow T \ast P + T & \Rightarrow P \ast P + T
\end{align*}
\]
Derivations

A *derivation* step is to replace the lhs of a production by its rhs.

**Example:**

<table>
<thead>
<tr>
<th>Step</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$E \rightarrow E + T$</td>
</tr>
<tr>
<td>2.</td>
<td>$E \rightarrow T$</td>
</tr>
<tr>
<td>3.</td>
<td>$T \rightarrow T * P$</td>
</tr>
<tr>
<td>4.</td>
<td>$T \rightarrow P$</td>
</tr>
<tr>
<td>5.</td>
<td>$P \rightarrow \text{id}$</td>
</tr>
</tbody>
</table>

$E \quad \rightarrow \quad E + T \quad \Rightarrow \quad T + T$

$E \quad \Rightarrow \quad T \quad \rightarrow \quad T * P + T \quad \Rightarrow \quad P * P + T$

$P \quad \Rightarrow \quad \text{id} * P + T$
Derivations

A *derivation* step is to replace the lhs of a production by its rhs.

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>E → E + T</th>
<th>E → T</th>
<th>T → T * P</th>
<th>T → P</th>
<th>P → id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
E & \Rightarrow E + T \quad \Rightarrow T + T \\
& \Rightarrow T * P + T \quad \Rightarrow P * P + T \\
& \Rightarrow id * P + T \quad \Rightarrow id * id + T
\end{align*}
\]
Derivations

A derivation step is to replace the lhs of a production by its rhs.

**Example:**

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T \ast P$
4. $T \rightarrow P$
5. $P \rightarrow id$

---

Let $E$ be the expression $E + T$.

- $E \rightarrow E + T$
- $E + T \rightarrow T + T$
- $T + T \rightarrow T \ast P + T$
- $T \ast P + T \rightarrow P \ast P + T$
- $P \ast P + T \rightarrow id \ast P + T$
- $id \ast P + T \rightarrow id \ast id + T$
- $id \ast id + T \rightarrow id \ast id + P$
Derivations

A *derivation* step is to replace the lhs of a production by its rhs.

**Example:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$E \rightarrow E + T$</td>
</tr>
<tr>
<td>2.</td>
<td>$E \rightarrow T$</td>
</tr>
<tr>
<td>3.</td>
<td>$T \rightarrow T \ast P$</td>
</tr>
<tr>
<td>4.</td>
<td>$T \rightarrow P$</td>
</tr>
<tr>
<td>5.</td>
<td>$P \rightarrow id$</td>
</tr>
</tbody>
</table>

$E$  
$\Rightarrow$  $E + T$  
$\Rightarrow$  $T + T$  
$\Rightarrow$  $T \ast P + T$  
$\Rightarrow$  $P \ast P + T$  
$\Rightarrow$  $id \ast P + T$  
$\Rightarrow$  $id \ast id + T$  
$\Rightarrow$  $id \ast id + P$  
$\Rightarrow$  $id \ast id + id$
Derivations

A *derivation* step is to replace the lhs of a production by its rhs.

**Example:**

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T \times P$
4. $T \rightarrow P$
5. $P \rightarrow id$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$E \rightarrow E + T$</td>
</tr>
<tr>
<td>2.</td>
<td>$E \rightarrow T$</td>
</tr>
<tr>
<td>3.</td>
<td>$T \rightarrow T \times P$</td>
</tr>
<tr>
<td>4.</td>
<td>$T \rightarrow P$</td>
</tr>
<tr>
<td>5.</td>
<td>$P \rightarrow id$</td>
</tr>
</tbody>
</table>

The set of sentences *derived* from the start symbol $S$ of a grammar $G$ defines the *language of the grammar*, denoted $L(G)$:

$$L(G) = \{ x \in V_t^* \mid S \Rightarrow x \}.$$
Derivations

A *derivation* step is to replace the lhs of a production by its rhs.

**Example:**

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \ast P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow \text{id} \)

<table>
<thead>
<tr>
<th>Step</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( E \Rightarrow E + T )</td>
</tr>
<tr>
<td>2</td>
<td>( E \Rightarrow T + T )</td>
</tr>
<tr>
<td>3</td>
<td>( T \Rightarrow T \ast P + T )</td>
</tr>
<tr>
<td>4</td>
<td>( P \Rightarrow P \ast P + T )</td>
</tr>
<tr>
<td>5</td>
<td>( \text{id} \Rightarrow \text{id} \ast \text{id} + T )</td>
</tr>
</tbody>
</table>

The set of sentences *derived* from the start symbol \( S \) of a grammar \( G \) defines the **language of the grammar**, denoted \( L(G) \):

\[
L(G) = \{ x \in V_t^* \mid S \Rightarrow x \}. 
\]

**Note:** Derivations are defined with respect to *proper* productions. They don’t work with grammars in EBNF form. While the rules in an EBNF grammar are of the form "lhs \( \rightarrow \) rhs", they are not proper productions.
Parse Tree

A.k.a. concrete syntax tree — A tree representation of complete derivations, i.e., from the start symbol to a sentence.
Parse Tree

A.k.a. concrete syntax tree — A tree representation of complete derivations, i.e., from the start symbol to a sentence.

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \ast P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow \text{id} \)

\[
\begin{align*}
E & \xrightarrow{1} E + T & \xrightarrow{2} T + T & \xrightarrow{3} T \ast P + T \\
& \xrightarrow{4} P \ast P + T & \xrightarrow{5} \text{id} \ast P + T & \xrightarrow{5} \text{id} \ast \text{id} + T \\
& \xrightarrow{4} \text{id} \ast \text{id} + P & \xrightarrow{5} \text{id} \ast \text{id} + \text{id}
\end{align*}
\]

- root node ⇔ start symbol
- leaf nodes ⇔ terminals
- non-leaf nodes ⇔ nonterminals
- non-leaf nodes with children ⇔ productions
Multiple Derivations

The same sentence often can be derived by different sequences of productions:

*Example:*

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \ast P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow \text{id} \)
Multiple Derivations

The same sentence often can be derived by different sequences of productions:

**Example:**

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \ast P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow \text{id} \)

Three different derivations:

1. \( E \xrightarrow{1} E + T \xrightarrow{2} T + T \xrightarrow{3} T \ast P + T \xrightarrow{4} P \ast P + T \)
   \( \xrightarrow{5} \text{id} \ast P + T \xrightarrow{5} \text{id} \ast \text{id} + T \xrightarrow{4} \text{id} \ast \text{id} + P \xrightarrow{5} \text{id} \ast \text{id} + \text{id} \)
Multiple Derivations

The same sentence often can be derived by different sequences of productions:

Example:

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \ast P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow \text{id} \)

Three different derivations:

1. \( E \overset{1}{\Rightarrow} E + T \overset{2}{\Rightarrow} T + T \overset{3}{\Rightarrow} T \ast P + T \overset{4}{\Rightarrow} P \ast P + T \)
   \( \overset{5}{\Rightarrow} \text{id} \ast P + T \overset{5}{\Rightarrow} \text{id} \ast \text{id} + T \overset{4}{\Rightarrow} \text{id} \ast \text{id} + P \overset{5}{\Rightarrow} \text{id} \ast \text{id} + \text{id} \)

2. \( E \overset{1}{\Rightarrow} E + T \overset{2}{\Rightarrow} T + T \overset{3}{\Rightarrow} T \ast P + T \overset{4}{\Rightarrow} P \ast P + T \)
   \( \overset{4}{\Rightarrow} P \ast P + P \overset{5}{\Rightarrow} P \ast P + \text{id} \overset{5}{\Rightarrow} P \ast \text{id} + \text{id} \overset{5}{\Rightarrow} \text{id} \ast \text{id} + \text{id} \)
Multiple Derivations

The same sentence often can be derived by different sequences of productions:

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>1.</th>
<th>2.</th>
<th>3.</th>
<th>4.</th>
<th>5.</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>(\rightarrow)</td>
<td>E + T</td>
<td>T + T</td>
<td>T * P + T</td>
<td>P * P + T</td>
</tr>
<tr>
<td></td>
<td>(\rightarrow)</td>
<td>T + T</td>
<td>T * P + T</td>
<td>P * P + T</td>
<td>id * id + P</td>
</tr>
<tr>
<td></td>
<td>(\rightarrow)</td>
<td>P * P + P</td>
<td>P * P + id</td>
<td>P * id + id</td>
<td>id * id + id</td>
</tr>
<tr>
<td></td>
<td>(\rightarrow)</td>
<td>P * P + P</td>
<td>P * P + id</td>
<td>P * id + id</td>
<td>id * id + id</td>
</tr>
<tr>
<td></td>
<td>(\rightarrow)</td>
<td>P * P + P</td>
<td>P * P + id</td>
<td>P * id + id</td>
<td>id * id + id</td>
</tr>
<tr>
<td>P</td>
<td>(\rightarrow)</td>
<td>P * P + P</td>
<td>P * P + id</td>
<td>P * id + id</td>
<td>id * id + id</td>
</tr>
</tbody>
</table>

Three different derivations:

1. \(E \Rightarrow E + T \Rightarrow T + T \Rightarrow T * P + T \Rightarrow P * P + T \Rightarrow id * P + T \Rightarrow id * id + T \Rightarrow id * id + P \Rightarrow id * id + id\)

2. \(E \Rightarrow E + T \Rightarrow T + T \Rightarrow T * P + T \Rightarrow P * P + T \Rightarrow P * P + P \Rightarrow P * P + id \Rightarrow P * id + id \Rightarrow id * id + id\)

3. \(E \Rightarrow E + T \Rightarrow E + P \Rightarrow E + id \Rightarrow T + id \Rightarrow T * P + id \Rightarrow P * P + id \Rightarrow id * P + id \Rightarrow id * id + id\)
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T * P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow \text{id} \)

\[
\begin{align*}
E & \Rightarrow E + T \\
  & \Rightarrow T + T \\
  & \Rightarrow T * P + T \\
  & \Rightarrow P * P + T \\
  & \Rightarrow \text{id} * P + T \\
  & \Rightarrow \text{id} * \text{id} + T \\
  & \Rightarrow \text{id} * \text{id} + P \\
  & \Rightarrow \text{id} * \text{id} + \text{id}
\end{align*}
\]
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T * P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow id \)

\[
\begin{align*}
E & \quad \Rightarrow \quad E + T \quad \Rightarrow \quad T + T \\
\Rightarrow \quad T * P + T \quad \Rightarrow \quad P * P + T \\
\Rightarrow \quad id * P + T \quad \Rightarrow \quad id * id + T \\
\Rightarrow \quad id * id + P \quad \Rightarrow \quad id * id + id
\end{align*}
\]
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * P$
4. $T \rightarrow P$
5. $P \rightarrow id$

$$E \quad \frac{1}{\Rightarrow} \quad E + T \quad \frac{2}{\Rightarrow} \quad T + T$$
$$\quad \frac{3}{\Rightarrow} \quad T * P + T \quad \frac{4}{\Rightarrow} \quad P * P + T$$
$$\quad \frac{5}{\Rightarrow} \quad id * P + T \quad \frac{5}{\Rightarrow} \quad id * id + T$$
$$\quad \frac{4}{\Rightarrow} \quad id * id + P \quad \frac{5}{\Rightarrow} \quad id * id + id$$
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T * P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow id \)

\[
\begin{align*}
E & \quad \Rightarrow \quad E + T \\
& \quad \Rightarrow \quad T + T \\
& \quad \Rightarrow \quad T * P + T \\
& \quad \Rightarrow \quad P * P + T \\
& \quad \Rightarrow \quad id * P + T \\
& \quad \Rightarrow \quad id * id + T \\
& \quad \Rightarrow \quad id * id + id
\end{align*}
\]
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \ast P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow \text{id} \)

\[
\begin{align*}
E & \Rightarrow_1 E + T & \Rightarrow_2 T + T \\
& \Rightarrow_3 T \ast P + T & \Rightarrow_4 P \ast P + T \\
& \Rightarrow_5 \text{id} \ast P + T & \Rightarrow_5 \text{id} \ast \text{id} + T \\
& \Rightarrow_4 \text{id} \ast \text{id} + P & \Rightarrow_5 \text{id} \ast \text{id} + \text{id}
\end{align*}
\]
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T * P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow \text{id} \)

\[
\begin{align*}
E & \Rightarrow E + T \quad \Rightarrow T + T \\
& \Rightarrow T * P + T \quad \Rightarrow P * P + T \\
& \Rightarrow \text{id} * P + T \quad \Rightarrow \text{id} * \text{id} + T \\
& \Rightarrow \text{id} * \text{id} + \text{P} \quad \Rightarrow \text{id} * \text{id} + \text{id}
\end{align*}
\]
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * P$
4. $T \rightarrow P$
5. $P \rightarrow \text{id}$

$E \rightarrow E + T \rightarrow T + T \rightarrow T * P + T \rightarrow \text{id} * P + T \rightarrow \text{id} * \text{id} + \text{T}$

Diagram:

```
      E
     /\  \\
    +   T
   /    \\
  T * P
 /    \\
P     id2
   /    \\
  id1
```
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \ast P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow \text{id} \)

\[
\begin{align*}
E & \Rightarrow E + T \Rightarrow T + T \\
& \Rightarrow T \ast P + T \Rightarrow P \ast P + T \\
& \Rightarrow \text{id} \ast P + T \Rightarrow \text{id} \ast \text{id} + T \\
& \Rightarrow \text{id} \ast \text{id} + P \Rightarrow \text{id} \ast \text{id} + \text{id}
\end{align*}
\]
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T * P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow id \)

\[
\begin{align*}
E & \xrightarrow{1} E + T \xrightarrow{4} E + P \\
& \xrightarrow{5} E + id \xrightarrow{2} T + id \\
& \xrightarrow{5} T * P + id \xrightarrow{3} T * id + id \\
& \xrightarrow{4} P * id + id \xrightarrow{5} id * id + id
\end{align*}
\]
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T * P \)
4. \( T \rightarrow P \)
5. \( P \rightarrow id \)

\[
\begin{align*}
E & \xrightarrow{1} E + T \quad \xrightarrow{4} E + P \\
& \xrightarrow{5} E + id \quad \xrightarrow{2} T + id \\
& \xrightarrow{5} T * P + id \quad \xrightarrow{3} T * id + id \\
& \xrightarrow{4} P * id + id \quad \xrightarrow{5} id * id + id
\end{align*}
\]
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * P$
4. $T \rightarrow P$
5. $P \rightarrow id$

$$
\begin{array}{c}
E & \Rightarrow & E + T & \Rightarrow & E + P \\
\Rightarrow & E + id & \Rightarrow & T + id \\
\Rightarrow & T * P + id & \Rightarrow & T * id + id \\
\Rightarrow & P * id + id & \Rightarrow & id * id + id
\end{array}
$$
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the *same* parse tree.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \rightarrow E + T )</td>
<td>( E \rightarrow E + T \rightarrow E + T )</td>
</tr>
<tr>
<td>( E \rightarrow T )</td>
<td>( E \rightarrow T \rightarrow T )</td>
</tr>
<tr>
<td>( T \rightarrow T * P )</td>
<td>( T \rightarrow T * P \rightarrow T * P )</td>
</tr>
<tr>
<td>( T \rightarrow P )</td>
<td>( T \rightarrow P \rightarrow P )</td>
</tr>
<tr>
<td>( P \rightarrow id )</td>
<td>( P \rightarrow id \rightarrow id )</td>
</tr>
</tbody>
</table>

E \( 1 \rightarrow E + T \) \( 4 \rightarrow E + P \) \( 5 \rightarrow E + id \) \( 2 \rightarrow T + id \) \( 5 \rightarrow T * P + id \) \( 3 \rightarrow T * id + id \) \( 4 \rightarrow P * id + id \) \( 5 \rightarrow id * id + id \)
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T \ast P$
4. $T \rightarrow P$
5. $P \rightarrow id$

\[
\begin{align*}
E & \Rightarrow E + T & \Rightarrow E + P \\
& \Rightarrow E + id & \Rightarrow T + id \\
& \Rightarrow T \ast P + id & \Rightarrow T \ast id + id \\
& \Rightarrow P \ast id + id & \Rightarrow id \ast id + id
\end{align*}
\]
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the *same* parse tree.

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * P$
4. $T \rightarrow P$
5. $P \rightarrow id$

\[
\begin{array}{l}
E \ \overset{1}{\Rightarrow} \ E + T \ \overset{4}{\Rightarrow} \ E + P \\
\ \ \ \overset{5}{\Rightarrow} \ E + id \ \overset{2}{\Rightarrow} \ T + id \\
\ \ \ \overset{5}{\Rightarrow} \ T * P + id \ \overset{3}{\Rightarrow} \ T * id + id \\
\ \ \ \overset{4}{\Rightarrow} \ P * id + id \ \overset{5}{\Rightarrow} \ id * id + id
\end{array}
\]
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the *same* parse tree.

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * P$
4. $T \rightarrow P$
5. $P \rightarrow id$

E \begin{array}{c}{1} \Rightarrow E + T \\ {5} \Rightarrow E + id \\ {5} \Rightarrow T * P + id \\ {4} \Rightarrow P * id + id \\ {5} \Rightarrow id * id + id \end{array} \begin{array}{c} \Rightarrow E + P \\ \Rightarrow T + id \\ \Rightarrow T * id + id \end{array}
Multiple Derivations

For unambiguous grammars, different derivations correspond to different ways of building up the same parse tree.

1. \[ E \rightarrow E + T \]
2. \[ E \rightarrow T \]
3. \[ T \rightarrow T * P \]
4. \[ T \rightarrow P \]
5. \[ P \rightarrow \text{id} \]

\[
\begin{align*}
E & \xrightarrow{1} E + T \xrightarrow{4} E + P \\
& \xrightarrow{5} E + \text{id} \xrightarrow{2} T + \text{id} \\
& \xrightarrow{5} T * P + \text{id} \xrightarrow{3} T * \text{id} + \text{id} \\
& \xrightarrow{4} P * \text{id} + \text{id} \xrightarrow{5} \text{id} * \text{id} + \text{id}
\end{align*}
\]
Left-Most and Right-Most Derivations

Only two special forms of derivations are of interests to a compiler:

- **Left-most derivation \((\Rightarrow_{lm})\) —**
  
  Always choose the left-most nonterminal to expand

  \[
  E \xrightarrow{1} E + T \xrightarrow{2} T + T \xrightarrow{3} T \ast P + T \xrightarrow{4} P \ast P + T \xrightarrow{5} id \ast P + T
  \]

  \[
  \xrightarrow{5} id \ast id + T \xrightarrow{4} id \ast id + P \xrightarrow{5} id \ast id + id
  \]
Left-Most and Right-Most Derivations

Only two special forms of derivations are of interests to a compiler:

► **Left-most derivation** \( \Rightarrow_{lm} \) —
Always choose the left-most nonterminal to expand

\[
\begin{align*}
E & \Rightarrow_1 E + T \\
E + T & \Rightarrow_2 T + T \\
T + T & \Rightarrow_3 T \ast P + T \\
P \ast P + T & \Rightarrow_4 P \ast P + T \\
\text{id} \ast P + T & \Rightarrow_5 \text{id} \ast \text{id} + id \\
\end{align*}
\]

► **Right-most derivation** \( \Rightarrow_{rm} \) —
Always choose the right-most nonterminal to expand

\[
\begin{align*}
E & \Rightarrow_1 E + T \\
E + T & \Rightarrow_4 E + P \\
E + P & \Rightarrow_5 E + \text{id} \\
E + \text{id} & \Rightarrow_2 T + \text{id} \\
T + \text{id} & \Rightarrow_5 T \ast \text{P} + \text{id} \\
T \ast \text{id} + \text{id} & \Rightarrow_3 T \ast \text{id} + \text{id} \\
P \ast \text{id} + \text{id} & \Rightarrow_4 P \ast \text{id} + \text{id} \\
\text{id} \ast \text{id} + \text{id} & \Rightarrow_5 \text{id} \ast \text{id} + \text{id}
\end{align*}
\]
Ambiguity in Grammars

A grammar is said to be *ambiguous* if it can derive some sentence with more than one parse tree.
Ambiguity in Grammars

A grammar is said to be *ambiguous* if it can derive some sentence with more than one parse tree.

- Different parse trees represent different program structures; implying different program semantics.
Ambiguity in Grammars

A grammar is said to be *ambiguous* if it can derive some sentence with more than one parse tree.

- Different parse trees represent different program structures; implying different program semantics.
- Parsers can not be built with an ambiguous grammar!
Ambiguity in Grammars

A grammar is said to be *ambiguous* if it can derive some sentence with more than one parse tree.

- Different parse trees represent different program structures; implying different program semantics.
- Parsers cannot be built with an ambiguous grammar!

Approaches for eliminating ambiguity:
Ambiguity in Grammars

A grammar is said to be *ambiguous* if it can derive some sentence with more than one parse tree.

- Different parse trees represent different program structures; implying different program semantics.
- Parsers can not be built with an ambiguous grammar!

Approaches for eliminating ambiguity:

- **Additional semantic information** — Provide the info as supplement to the grammar.
Ambiguity in Grammars

A grammar is said to be *ambiguous* if it can derive some sentence with more than one parse tree.

- Different parse trees represent different program structures; implying different program semantics.
- Parsers can not be built with an ambiguous grammar!

Approaches for eliminating ambiguity:

- **Additional semantic information** —
  Provide the info as supplement to the grammar.
- **Grammar rewriting** —
  Encode additional semantic information into a grammar.
Ambiguity in Grammars

\[
E \rightarrow E + T \mid E \ast T \\
T \rightarrow \text{id}
\]

*Input:* \( \text{id1} + \text{id2} \ast \text{id3} \)
Ambiguity in Grammars

\[
E \rightarrow E \, + \, T \mid E \, \ast \, T \\
T \rightarrow \text{id}
\]

**Input:** \(id1 + id2 \ast id3\)

- Two parse trees for the same input:
Ambiguity in Grammars

\[
E \rightarrow E + T \mid E * T \\
T \rightarrow id
\]

**Input:** id1 + id2 * id3

- Two parse trees for the same input:

  - Each tree has its own left-most- and right-most-derivations.
Ambiguity in Grammars

\[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \]
\[ \mid \text{if } E \text{ then } S \]
\[ \mid \text{other} \]

*Input:* if \( E_1 \) then if \( E_2 \) then \( S_1 \) else \( S_2 \)
Ambiguity in Grammars

\[
S \to \text{if } E \text{ then } S \text{ else } S \\
| \text{if } E \text{ then } S \\
| \text{other}
\]

Input: \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2

▶ Two possible parse trees:
Ambiguity in Grammars

\[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \]
\[ \mid \text{if } E \text{ then } S \]
\[ \mid \text{other} \]

*Input:* if \( E_1 \) then if \( E_2 \) then \( S_1 \) else \( S_2 \)

- Two possible parse trees:

  - Implied interpretations:
    - if \( E_1 \) then \{ if \( E_2 \) then \( S_1 \) else \( S_2 \) \}
    - if \( E_1 \) then \{ if \( E_2 \) then \( S_1 \) \} else \( S_2 \)
Grammar Equivalence

Two grammars that define the same language are said to be equivalent to each other.

Examples:

\[ G_1: S \rightarrow S \times \mid \epsilon \]
\[ G_2: S \rightarrow \times S \mid \epsilon \]

Both \( G_1 \) and \( G_2 \) define the set \( \{x^*\} \).
Grammar Equivalence

Two grammars that define the same language are said to be equivalent to each other.

Examples:

\[
\begin{array}{l}
G_1: \quad S \rightarrow S \times \mid \epsilon \\
G_2: \quad S \rightarrow \times S \mid \epsilon
\end{array}
\]

Both \(G_1\) and \(G_2\) define the set \(\{x^*\}\).

\[
\begin{array}{l}
G_3: \quad E \rightarrow E + E \mid E - E \mid \text{id} \\
G_4: \quad E \rightarrow E + \text{id} \mid E - \text{id} \mid \text{id}
\end{array}
\]

Both \(G_3\) and \(G_4\) define expressions with \(+\) and \(-\). \(G_3\) is ambiguous, \(G_4\) is not.
Grammar Equivalence

Two grammars that define the same language are said to be equivalent to each other.

Examples:

\[
G_1: \quad S \rightarrow S \times | \epsilon \\
G_2: \quad S \rightarrow \times S | \epsilon \\
\]

Both \(G_1\) and \(G_2\) define the set \(\{x^*\}\).

\[
G_3: \quad E \rightarrow E + E | E - E | \text{id} \\
G_4: \quad E \rightarrow E + \text{id} | E - \text{id} | \text{id} \\
\]

Both \(G_3\) and \(G_4\) define expressions with + and −. \(G_3\) is ambiguous, \(G_4\) is not.

- Grammar equivalence can be taken advantage of in dealing with ambiguity in programming languages’ grammar.
- In particular, sometimes we can encode semantic information into a grammar to eliminate ambiguity.
Encode Semantics in Grammar

Example 1. Encode operator associativity.

- Ambiguous grammar:

1. $E \rightarrow E + E$
2. $E \rightarrow E - E$
3. $E \rightarrow id$
Encode Semantics in Grammar

Example 1. Encode operator associativity.

▷ Ambiguous grammar:

1. $E \rightarrow E + E$
2. $E \rightarrow E - E$
3. $E \rightarrow \text{id}$

▷ Equivalent, unambiguous grammars:

Implying left associativity:

1. $E \rightarrow E + \text{id}$
2. $E \rightarrow E - \text{id}$
3. $E \rightarrow \text{id}$
**Example 1.** Encode operator associativity.

- **Ambiguous grammar:**
  1. \( E \rightarrow E + E \)
  2. \( E \rightarrow E - E \)
  3. \( E \rightarrow \text{id} \)

- **Equivalent, unambiguous grammars:**
  - Implying left associativity:
    1. \( E \rightarrow E + \text{id} \)
    2. \( E \rightarrow E - \text{id} \)
    3. \( E \rightarrow \text{id} \)
  - Implying right associativity:
    1. \( E \rightarrow \text{id} + E \)
    2. \( E \rightarrow \text{id} - E \)
    3. \( E \rightarrow \text{id} \)
Encode Semantics in Grammar

Example 1. Encode operator associativity.

- Ambiguous grammar:

  1. \[ E \rightarrow E + E \]
  2. \[ E \rightarrow E - E \]
  3. \[ E \rightarrow \text{id} \]

- Equivalent, unambiguous grammars:

  Implying left associativity:

  1. \[ E \rightarrow E + \text{id} \]
  2. \[ E \rightarrow E - \text{id} \]
  3. \[ E \rightarrow \text{id} \]

  Implying right associativity:

  1. \[ E \rightarrow \text{id} + E \]
  2. \[ E \rightarrow \text{id} - E \]
  3. \[ E \rightarrow \text{id} \]

General Rules:

- Left-recursion implies left associativity.
- Right-recursion implies right associativity.
Encode Semantics in Grammar

Example 1. (cont.)

Input: $id_1 + id_2 - id_3$
Encode Semantics in Grammar

Example 1. (cont.)

Input:

id1 + id2 − id3

Original grammar – multiple parse trees:
Encode Semantics in Grammar

Example 1. (cont.)

Input: \( \text{id1} + \text{id2} - \text{id3} \)

Original grammar – multiple parse trees:

New grammars – unique parse tree:
Encode Semantics in Grammar

Example 2. Encode operator precedence.

- Ambiguous grammar:

1. $E \rightarrow E + E$
2. $E \rightarrow E \ast E$
3. $E \rightarrow \text{id}$
Encode Semantics in Grammar

Example 2. Encode operator precedence.

- Ambiguous grammar:
  1. $E \rightarrow E + E$
  2. $E \rightarrow E * E$
  3. $E \rightarrow \text{id}$

- Equivalent, unambiguous grammars:
  * has higher precedence:
  1. $E \rightarrow E + T$
  2. $E \rightarrow T$
  3. $T \rightarrow T * \text{id}$
  4. $T \rightarrow \text{id}$
Encode Semantics in Grammar

Example 2. Encode operator precedence.

- Ambiguous grammar:

  1. $E \rightarrow E + E$
  2. $E \rightarrow E \ast E$
  3. $E \rightarrow \text{id}$

- Equivalent, unambiguous grammars:

  * has higher precedence:

    1. $E \rightarrow E + T$
    2. $E \rightarrow T$
    3. $T \rightarrow T \ast \text{id}$
    4. $T \rightarrow \text{id}$

  + has higher precedence:

    1. $E \rightarrow E \ast T$
    2. $E \rightarrow T$
    3. $T \rightarrow T + \text{id}$
    4. $T \rightarrow \text{id}$
Encode Semantics in Grammar

Example 2. Encode operator precedence.

▶ Ambiguous grammar:

1. \( E \rightarrow E + E \)
2. \( E \rightarrow E \ast E \)
3. \( E \rightarrow \text{id} \)

▶ Equivalent, unambiguous grammars:

* has higher precedence:

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \ast \text{id} \)
4. \( T \rightarrow \text{id} \)

+ has higher precedence:

1. \( E \rightarrow E \ast T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T + \text{id} \)
4. \( T \rightarrow \text{id} \)

General Rules:

▶ Introduce a new nonterminal for each level of precedence.
▶ Place the higher-precedence operator in the “lower” productions.
Encode Semantics in Grammar

Example 2. (cont.)

Input: \( id1 + id2 \ast id3 \)
Encode Semantics in Grammar

**Example 2.** (cont.)

*Input:* 

id1 + id2 * id3

Original grammar – multiple parse trees:
Encode Semantics in Grammar

Example 2. (cont.)

Input:

Original grammar – multiple parse trees:

New grammars – unique parse tree:

id1 + id2 * id3

id1 + (id2 * id3)  (id1 + id2) * id3
Example 3. Encode else clause binding.

- Ambiguous grammar:

1. $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
2. $S \rightarrow \text{if } E \text{ then } S$
3. $S \rightarrow \text{other}$
Encode Semantics in Grammar

*Example 3.* Encode else clause binding.

- **Ambiguous grammar:**
  1. \( S \rightarrow \text{if } E \text{ then } S \text{ else } S \)
  2. \( S \rightarrow \text{if } E \text{ then } S \)
  3. \( S \rightarrow \text{other} \)

- **Equivalent, unambiguous grammar:**
  1. \( S \rightarrow M \)
  2. \( S \rightarrow U \)
  3. \( M \rightarrow \text{if } E \text{ then } M \text{ else } M \)
  4. \( M \rightarrow \text{other} \)
  5. \( U \rightarrow \text{if } E \text{ then } M \text{ else } U \)
  6. \( U \rightarrow \text{if } E \text{ then } S \)

\( M \) represents matched if stmt; 
\( U \) represents unmatched if stmt.
Encode Semantics in Grammar

Example 3. (cont.)

Input: \( \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2 \)

Original: multiple parse trees

![Diagram of parse trees]

if \( E_1 \) then \{if \( E_2 \) then \( S_1 \) else \( S_2 \)\}

if \( E_1 \) then \{if \( E_2 \) then \( S_1 \)\} else \( S_2 \)
Encode Semantics in Grammar

Example 3. (cont.)

Input: \[ \text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2 \]

Original: multiple parse trees

New: unique parse tree

\[
\begin{align*}
S & \rightarrow \text{if } E_1 \text{ then } S \\
& \quad \rightarrow \text{if } E_2 \text{ then } S_1 \text{ else } S_2 \\
& \quad \quad \rightarrow \text{if } E_1 \text{ then } \{ \text{if } E_2 \text{ then } S_1 \text{ else } S_2 \}
\end{align*}
\]

\[
\begin{align*}
S & \rightarrow \text{if } E_1 \text{ then } S \\
& \quad \rightarrow \text{if } E_2 \text{ then } M \text{ else } M \\
& \quad \quad \rightarrow \text{if } E_1 \text{ then } \{ \text{if } E_2 \text{ then } S_1 \text{ else } S_2 \}
\end{align*}
\]
Encode Semantics in Grammar

Details are Important. Slight change in grammar can cause major property change:

► Unambiguous:

1. \( S \rightarrow M \)
2. \( S \rightarrow U \)
3. \( M \rightarrow \text{if } E \text{ then } M \text{ else } M \)
4. \( M \rightarrow \text{other} \)
5. \( U \rightarrow \text{if } E \text{ then } M \text{ else } U \)
6. \( U \rightarrow \text{if } E \text{ then } S \)
Encode Semantics in Grammar

Details are Important. Slight change in grammar can cause major property change:

- Unambiguous:

  1. \( S \rightarrow M \)
  2. \( S \rightarrow U \)
  3. \( M \rightarrow \text{if } E \text{ then } M \text{ else } M \)
  4. \( M \rightarrow \text{other} \)
  5. \( U \rightarrow \text{if } E \text{ then } M \text{ else } U \)
  6. \( U \rightarrow \text{if } E \text{ then } S \)

- Ambiguous:

  1. \( S \rightarrow M \)
  2. \( S \rightarrow U \)
  3. \( M \rightarrow \text{if } E \text{ then } M \text{ else } S \)
  4. \( M \rightarrow \text{other} \)
  5. \( U \rightarrow \text{if } E \text{ then } S \)

  1. \( S \rightarrow \text{if } E \text{ then } S \)
  2. \( S \rightarrow M \)
  3. \( M \rightarrow \text{if } E \text{ then } M \text{ else } S \)
  4. \( M \rightarrow \text{other} \)
Non-Regular and Non-Context-Free Examples

- \( L_1 = \{ [i]^i \mid i \geq 1 \} \) — non-regular
Non-Regular and Non-Context-Free Examples

- $L_1 = \{ [i]^i \mid i \geq 1 \}$ — non-regular

**Implication:** The syntax of “nested block comments” cannot be described by RE.
Non-Regular and Non-Context-Free Examples

- $L_1 = \{ [i]^i \mid i \geq 1 \}$ — non-regular
  
  **Implication:** The syntax of “nested block comments” cannot be described by RE.

- $L_2 = \{ wcw \mid w \text{ is in } (a|b)^* \}$ — non-CF
Non-Regular and Non-Context-Free Examples

- \( L_1 = \{ \,[i]^i \,|\, i \geq 1 \} \) — non-regular

  Implication: The syntax of “nested block comments” cannot be described by RE.

- \( L_2 = \{ wcw \,|\, w \text{ is in } (a|b)^* \} \) — non-CF

  Implication: The checking of “identifiers are declared before their use” cannot be handled by CFG.
Non-Regular and Non-Context-Free Examples

- \( L_1 = \{ [i]^i \mid i \geq 1 \} \) — non-regular

  **Implication:** The syntax of “nested block comments” cannot be described by RE.

- \( L_2 = \{ wcw \mid w \text{ is in } (a|b)^* \} \) — non-CF

  **Implication:** The checking of “identifiers are declared before their use” cannot be handled by CFG.

- \( L_3 = \{ a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1 \} \) — non-CF
Non-Regular and Non-Context-Free Examples

- $L_1 = \{ [i]^i \mid i \geq 1 \}$ — non-regular
  
  *Implication:* The syntax of “nested block comments” cannot be described by RE.

- $L_2 = \{ wcw \mid w \text{ is in } (a|b)^* \}$ — non-CF
  
  *Implication:* The checking of “identifiers are declared before their use” cannot be handled by CFG.

- $L_3 = \{ a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1 \}$ — non-CF
  
  *Implication:* The checking of “the number of formal parameters in the declaration of a procedure agrees with the number of actual parameters in a use of the procedure” cannot be handled by CFG.
Summary

- Regular expression and context-free grammar are the formal-system of choice for program language’s token and syntax specifications.
Summary

- Regular expression and context-free grammar are the formal-system of choice for program language’s token and syntax specifications.
- In practice, alternative approaches are often used for dealing with more difficult cases, e.g. using “states” for token specifications, and providing external information for dealing with grammar ambiguity.
Summary

- Regular expression and context-free grammar are the formal-system of choice for program language’s token and syntax specifications.
- In practice, alternative approaches are often used for dealing with more difficult cases, e.g. using “states” for token specifications, and providing external information for dealing with grammar ambiguity.
- A grammar is ambiguous if it can derive some sentence with more than one parse tree. Ambiguous grammars are not suitable for parsing.
Summary

- Regular expression and context-free grammar are the formal-system of choice for program language’s token and syntax specifications.
- In practice, alternative approaches are often used for dealing with more difficult cases, e.g. using “states” for token specifications, and providing external information for dealing with grammar ambiguity.
- A grammar is ambiguous if it can derive some sentence with more than one parse tree. Ambiguous grammars are not suitable for parsing.
- It is possible to encode some semantic rules into a grammar.