How do we define a PL?

- Specifying a PL:
  - Syntax: the form (structure) of a program
  - Semantics: the meaning of a program
  - Pragmatics: the implementation of a PL
- Need a precise definition, without ambiguity
  - Given a program, there is only one unique interpretation
- Purpose:
  - For language designers: Convey the design principles of the language
  - For language implementers: Define precisely what is to be implemented
  - For language programmers: Describe the language that is to be used
- How to describe?
  - Natural language: ambiguous
  - Formal methods: very precise, used extensively (especially for syntax)

Scanning and Parsing

Lexical Structure: The structure of tokens (words)
scanning phase (lexical analysis): recognize tokens from characters

Syntactical Structure: The structure of programs
parsing phase (syntax analysis): determine the syntactic structure

AST: Abstract Syntax Tree

Scanning

- A scanner groups input characters into tokens

<table>
<thead>
<tr>
<th>input</th>
<th>token</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>equal</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>identifier</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>star</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>x = x * (acc+123)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>left-paren</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>identifier</td>
<td>acc</td>
<td></td>
</tr>
<tr>
<td>plus</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>right-paren</td>
<td>)</td>
<td></td>
</tr>
</tbody>
</table>

- Tokens are typically represented by numbers
**Parsing**

- Each time the parser needs a token, it sends a request to the scanner.
- The scanner reads as many characters from the input stream as necessary to construct a single token.
- When a single token is formed, the scanner is suspended and returns the token to the parser.
- The parser will repeatedly call the scanner to read all the tokens from the input stream.

**Tasks of a Scanner**

- A typical scanner:
  - recognizes the keywords of the language.
  - These are the reserved words that have a special meaning in the language, such as the words `class`, `while`, `if`, `...` in Java.
    - They cannot be redefined (as variables, etc).
  - Recognizes special characters, such as `(` and `)`, or groups of special characters, such as `:=` and `==`.
  - Recognizes identifiers.
    - Predefined identifiers: have initial meaning and allow redefinition.
      - Eg., `String`, `Object`, `Integer` in Java.
  - Recognizes constants: integers, reals, decimals, strings, etc.
  - Ignores whitespaces (tabs, blanks, etc) and comments.
  - Recognizes and processes special directives (such as the `#include` “file” directive in C) and macros.

**Scanner Generators**

- Input: a scanner specification.
  - Describes every token using Regular Expressions (REs).
    - Eg., the RE `[^a-zA-Z][^a-zA-Z-0-9]*` recognizes all identifiers with at least one alphanumeric letter whose first letter is lower-case alphabetic.
  - Handles whitespaces and resolve ambiguities.
- Output: the actual scanner.
- Scanner generators compile regular expressions into efficient programs (finite state machines).
  - Eg., lex, flex, JLex.

**Regular Expressions**

- Are a very convenient form of representing (possibly infinite) sets of strings, called regular sets.

<table>
<thead>
<tr>
<th>Name</th>
<th>RE</th>
</tr>
</thead>
<tbody>
<tr>
<td>epsilon</td>
<td>ε</td>
</tr>
<tr>
<td>symbol</td>
<td>a</td>
</tr>
<tr>
<td>concatenation</td>
<td>AB</td>
</tr>
<tr>
<td>selection</td>
<td>A</td>
</tr>
<tr>
<td>repetition</td>
<td>A*</td>
</tr>
</tbody>
</table>

Shortcuts: \( A^* \sim AA^* \)

A? = A|ε \[a-z]\sim (a|b|...|z)\]

- Eg. the RE `(a|b)*aaa` represents the infinite set \{“aa”, “aaa”, “bba”, “abaa”, ... \}
Examples

- **for-keyword** = for
- **letter** = [a-zA-Z]
- **digit** = [0-9]
- **identifier** = letter (letter | digit)*
- **sign** = + | - | ε
- **integer** = sign (0 | [1-9]digit*)
- **decimal** = integer . digit*
- **real** = (integer | decimal) E sign digit*

Disambiguation Rules

- **longest match (substring) rule**: from all tokens that match the input prefix, choose the one that matches the most characters
- **rule priority**: if more than one token has the longest match, choose the one listed first

Examples:
- for8 is it the for-keyword, the identifier “f”, the identifier “fo”, the identifier “for”, or the identifier “for8”?  
  *Use rule 1*: “for8” matches the most characters.
- for is it the for-keyword, the identifier “f”, the identifier “fo”, or the identifier “for”?  
  *Use rule 1 & 2*: the for-keyword and the “for” identifier have the longest match but the for-keyword is listed first.

Whitespaces

- Principle of longest match requires that tokens are separated by whitespace
- **Whitespace** (token delimiters):  
  - blanks, newlines, and tabs are ignored, except when they separate tokens  
  - an exception: FORTRAN  
    - DO 99 I = 1.10 (the same as DO99I=1.10)
- **Free-format language**: the format has no effect on the program structure  
  - Most languages are free format  
  - One exception: python  
  - Haskell uses Indention Syntax to denote structure (optional)  
    - greatly reduces the amount of junk syntax in the language, such as '{', '}', and ':'

Parser

- A parser recognizes sequences of tokens according to some grammar and generates Abstract Syntax Trees (ASTs)
- A **context-free grammar** (CFG) has  
  - a finite set of terminals (tokens)  
  - a finite set of nonterminals from which one is the start symbol  
  - and a finite set of productions of the form:  
    $$A \rightarrow X_1 X_2 \ldots X_n$$  
    where A is a nonterminal and each Xi is either a terminal or nonterminal symbol
Context-Free Grammars

- Example of grammar rules (productions):
  1. \textit{sentence} \rightarrow \textit{noun-phrase} \textit{verb-phrase}.
  2. \textit{noun-phrase} \rightarrow \textit{article noun}
  3. \textit{article} \rightarrow \text{a} | \text{the}
  4. \textit{noun} \rightarrow \textit{girl} | \textit{dog}
  5. \textit{verb-phrase} \rightarrow \textit{verb noun-phrase}
  6. \textit{verb} \rightarrow \textit{sees} | \textit{pets}

Languages produced by Grammars

Language: the set of strings (of terminals) that can be generated from the start symbol by derivation:

\begin{align*}
\text{sentence} & \Rightarrow \text{noun-phrase verb-phrase} . \\
& \Rightarrow \text{article noun verb-phrase} . \ (\text{rule 1}) \\
& \Rightarrow \text{the noun verb-phrase} . \ (\text{rule 2}) \\
& \Rightarrow \text{the girl verb-phrase} . \ (\text{rule 3}) \\
& \Rightarrow \text{the girl noun-phrase} . \ (\text{rule 4}) \\
& \Rightarrow \text{the girl verb noun-phrase} . \ (\text{rule 5}) \\
& \Rightarrow \text{the girl noun-phrase} . \ (\text{rule 6}) \\
& \Rightarrow \text{the girl sees noun-phrase} . \ (\text{rule 7}) \\
& \Rightarrow \text{the girl sees verb noun} . \ (\text{rule 8}) \\
& \Rightarrow \text{the girl sees noun} . \ (\text{rule 9}) \\
& \Rightarrow \text{the girl sees a noun} . \ (\text{rule 10}) \\
& \Rightarrow \text{the girl sees a dog} . \ (\text{rule 11})
\end{align*}

Context-Free Grammars (CFG)

- Context-Free Grammars (CFG)
  - Noam Chomsky, 1950s
  - They define context-free languages

  \begin{itemize}
    \item Four components:
      \begin{itemize}
        \item terminals, nonterminals, one start symbol, productions
      \end{itemize}
    \item Left-hand side of a production is always one single nonterminal:
      \begin{itemize}
        \item The nonterminal is replaced by the corresponding right-hand side, no matter where the nonterminal appears
        \item i.e., there is no context in such replacement/derivation
      \end{itemize}
  \end{itemize}

  Context-sensitive grammars
  - aX \rightarrow b
  - cX \rightarrow d

Why context-free?

Backus-Naur Form (BNF)

- A meta language used to describe CFG
- Introduced by John Backus/Peter Naur: for describing the syntax of Algol60
- BNF is formal and precise

\begin{align*}
\text{E} & \rightarrow \text{ID} \\
& \mid \text{NUM} \\
& \mid \text{E} \ast \text{E} \\
& \mid \text{E} / \text{E} \\
& \mid \text{E} + \text{E} \\
& \mid \text{E} - \text{E} \\
& \mid ( \text{E} ) \\
\text{ID} & \rightarrow \text{a} \mid \text{b} \mid \ldots \mid \text{z} \\
\text{NUM} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
Example #2

• An example that describes statements:
  S → if E then S else S  
  | begin S L
  | print E
L → end
  | ; S L
E → NUM = NUM
NUM → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Example #3

• Expressions with terms and factors:
  E → E + T
  | E - T
  | T
T → T * F
  | T / F
  | F
F → num
  | id

Derivations

• Notation:
  - terminals: t, s, ...
  - nonterminals: A, B, ...
  - symbol (terminal or nonterminal): X, Y, ...
  - sequence of symbols: a, b, ...

• Given a production:
  A ::= X₁ X₂ ... Xₙ
  the form aAb => aX₁ X₂ ... Xₙb is called a derivation

• eg, using the production  T ::= T * F  we get
  T / F + 1 - x => T * F / F + 1 - x

• Leftmost derivation: when you always expand the leftmost nonterminal in the sequence
• Rightmost derivation: ... rightmost nonterminal

Parse Tree

• Given the derivations used to parse an input sequence, a parse tree has
  - the start symbol as the root
  - the terminals of the input sequence as leaves
  - for each production  A ::= X₁ X₂ ... Xₙ used in a derivation, a node A with children X₁, X₂ ... Xₙ

From Example #3:

E => E + T
 => E + T * F
 => T + T * F
 => T + F * F
 => T + num * F
 => F + num * F
 => id + num * F
 => id + num * id
 => x + 2 * y
**Abstract Syntax Tree (AST)**

- **Abstract Syntax Tree:** Removes "unnecessary" terminals and nonterminals but still completely determines syntactic structure
- A parse tree is not an AST
  - A parse tree is tedious: all terminals and nonterminals in a derivation are included in the tree

```
E  
/  
T  
|  
T  
/  
F  
|  
F
```

id(x) + num(2) * id(y)

---

**AST Example**

```
If-statement

if
  ( expr )
stmt
else
stmt
```

---

**What is Parsing?**

- **Given a grammar and a token string:**
  - determine if the grammar can generate the token string
  - is the string a legal program in the language?

- **... or Better:**
  - construct a parse tree for the token string

---

**What’s significant about a parse tree?**

A parse tree gives a unique syntactic structure

- Leftmost vs rightmost derivation
- There is only one leftmost derivation and one rightmost derivation for a parse tree
Example

expr → expr + expr | expr * expr | ( expr ) | number
number → number digit | digit
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

What’s significant about parse tree?

A parse tree has a unique meaning
- It provides a basis for semantic analysis
- Syntax-directed semantics: semantics are attached to syntactic structure
  - eg, attribute grammars

Example

expr
  +
  expr
  number
digit
digit
digit

expr1 + expr2

expr.val = expr1.val + expr2.val

Language vs Grammar vs Parser

- Chomsky Hierarchy
<table>
<thead>
<tr>
<th>Grammar</th>
<th>Languages</th>
<th>Automata</th>
<th>Production rules (constraints)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 0</td>
<td>Recursively enumerable</td>
<td>Turing machine</td>
<td>(α → β) w.r.t. restrictions</td>
</tr>
<tr>
<td>Type 1</td>
<td>Context-sensitive</td>
<td>Linear-bounded non-deterministic Turing machine</td>
<td>αA → γβ</td>
</tr>
<tr>
<td>Type 3</td>
<td>Regular</td>
<td>Finite state automaton</td>
<td>A → a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A → aB</td>
</tr>
</tbody>
</table>

- A language can be described by multiple grammars, while a grammar defines only one language
- A grammar can be parsed by multiple parsers, while a parser accepts one grammar, thus one language
- Goal: design a language that allows a simple grammar and an efficient parser
  - Given a language, we should construct a grammar that allows fast parsing
  - Given a grammar, we should build an efficient parser

Ambiguity

- Ambiguous grammar: There can be multiple parse trees for the same token string
  - multiple leftmost derivations.
- Why is it bad?
  - Multiple interpretations => no unique meaning
**Ambiguity**

- Is this ambiguous?
  
  \[
  \begin{align*}
  \text{number} & \rightarrow \text{number} \text{ digit} \mid \text{digit} \\
  \text{digit} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
  \end{align*}
  \]

- How about this?
  
  \[
  \text{expr} \rightarrow \text{expr} \ - \ \text{expr} \mid \text{number}
  \]

- How to resolve ambiguity?
  - Rewrite the grammar to avoid ambiguity

**Example of Ambiguity: Precedence**

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} \ + \ \text{expr} \mid \text{expr} \ * \ \text{expr} \mid ( \ \text{expr} )
\end{align*}
\]

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 + 4 * 5</td>
<td>21</td>
</tr>
</tbody>
</table>

Two different parse trees for expression 3+4*5

**Example of Ambiguity: Associativity**

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} \ - \ \text{expr} \mid ( \ \text{expr} )
\end{align*}
\]

<table>
<thead>
<tr>
<th>expr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 - 2 - 1</td>
<td>2</td>
</tr>
<tr>
<td>5 - (2 - 1)</td>
<td>4</td>
</tr>
</tbody>
</table>

Two different parse trees for expression 5-2-1

**Eliminating Ambiguity for Precedence**

- Establish “precedence cascade”
  - use different structured names for different constructs, adding grammar rules
  - higher precedence = lower in cascade

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} \ + \ \text{expr} \mid \text{expr} \ * \ \text{expr} \mid ( \ \text{expr} ) \mid \text{number}
\end{align*}
\]

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} \ + \ \text{expr} \mid \text{term}
\end{align*}
\]

\[
\begin{align*}
\text{term} & \rightarrow \text{term} \ * \ \text{term} \mid ( \ \text{expr} ) \mid \text{number}
\end{align*}
\]
Eliminating Ambiguity for Associativity

- left-associativity: left-recursion

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} \cdot \text{expr} \mid (\text{expr}) \mid \text{number} \\
\text{term} & \rightarrow (\text{expr}) \mid \text{number}
\end{align*}
\]

- right-associativity: right-recursion

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} \cdot \text{expr} \mid a \mid b \mid c \\
\text{term} & \rightarrow \text{expr} \mid \text{term}
\end{align*}
\]

Putting all Together

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} \rightarrow \text{expr} \mid \text{expr} \cdot \text{expr} \mid (\text{expr}) \mid \text{number} \\
\text{number} & \rightarrow \text{number} \cdot \text{digit} \mid \text{digit} \\
\text{digit} & \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\end{align*}
\]

We want to make \texttt{minus} left-associative and \texttt{division} to have higher precedence than \texttt{minus}.

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} \rightarrow \text{term} \mid \text{term} \\
\text{term} & \rightarrow \text{term} / \text{factor} \mid \text{factor} \\
\text{factor} & \rightarrow (\text{expr}) \mid \text{number} \\
\text{digit} & \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\end{align*}
\]

Example of Ambiguity: dangling-else

\[
\begin{align*}
\text{stmt} & \rightarrow \text{if} (\text{expr}) \text{stmt} \\
& \quad \mid \text{if} (\text{expr}) \text{stmt} \text{else stmt} \\
& \quad \mid \text{other-stmt}
\end{align*}
\]

Two different parse trees for “\texttt{if (expr) if (expr) stmt else stmt}”

Eliminating Dangling-Else

\[
\begin{align*}
\text{stmt} & \rightarrow \text{matched_stmt} \\
& \quad \mid \text{unmatched_stmt} \\
\text{matched_stmt} & \rightarrow \text{if} (\text{expr}) \text{matched_stmt} \text{else matched_stmt} \\
& \quad \mid \text{other-stmt} \\
\text{unmatched_stmt} & \rightarrow \text{if} (\text{expr}) \text{stmt} \\
& \quad \mid \text{if} (\text{expr}) \text{matched_stmt} \text{else unmatched_stmt}
\end{align*}
\]

- “else” may occur in both matched_stmt and unmatched_stmt
- The statement before “else” must be a matched statement
  - Thus, the “else” will not match any “if” inside that statement, otherwise there will be more than one parse tree
EBNF

- Repetition { }
  \[ \text{number} \rightarrow \text{digit} \mid \text{number} \text{ digit} \]
  \[ \text{expr} \rightarrow \text{expr} \mid \text{term} \mid \text{term} \]
  \[ \text{term} \rightarrow \text{term} \mid \text{digit} \]

- Option [ ]
  \[ \text{signed-number} \rightarrow \text{sign} \mid \text{number} \]
  \[ \text{if-stmt} \rightarrow \text{if} \mid \text{else} \]

\[ \text{if-stmt} \rightarrow \text{if} \left( \text{expr} \right) \text{ stmt} \]
\[ \mid \text{if} \left( \text{expr} \right) \text{ stmt} \mid \text{else} \text{ stmt} \]

Syntax Diagrams

- Often drawn based on EBNF, not BNF
- Example: If-statement

```plaintext
if-stmt → if (expression) }
statement

if-stmt → if (expression) stmt [ else stmt ]
```

Parsing

- Parsing:
  - Determine if a grammar can generate a given token string
  - ... or better, construct a parse tree for the token string

- Two ways of constructing the parse tree
  - Top-down (from root towards leaves)
    - Can be constructed more easily by hand
  - Bottom-up (from leaves towards root)
    - Can handle a larger class of grammars
    - Parser generators tend to use bottom-up methods

Top-Down Parser

- Recursive-descent parser:
  - A special kind of top-down parser: single left-to-right scan, with one lookahead symbol
  - Backtracking (trial-and-error) may happen

- Predictive parser:
  - The lookahead symbol determines which production to apply, without backtracking

- Will use the example:

```plaintext
E ::= E + T | E - T | T
T ::= T * F | T / F | F
F ::= num | id
```
Top-down Parsing

- It starts from the start symbol of the grammar and applies derivations until the entire input string is derived
- Example that matches the input sequence id(x) + num(2) * id(y)
  \[
  \begin{align*}
  E & \rightarrow E + T \\
  & \Rightarrow E + T * F \\
  & \Rightarrow T + T * F \\
  & \Rightarrow T + F * F \\
  & \Rightarrow T + num * F \\
  & \Rightarrow F + num * F \\
  & \Rightarrow id + num * F \\
  & \Rightarrow id + num * id \\
  \end{align*}
  \]
  use \( E ::= E + T \)
  use \( E ::= E + T * F \)
  use \( E ::= T * F \)
  use \( T ::= F \)
  use \( T ::= num \)
  use \( F ::= F \)
  use \( F ::= id \)
  use \( F ::= id \)

- You may have more than one choice at each derivation step:
  - my have multiple nonterminals in each sequence
  - for each nonterminal in the sequence, may have many rules to choose from
- Wrong predictions will cause backtracking
  - need predictive parsing that never backtracks

Bottom-up Parsing

- It starts from the input string and uses derivations in the opposite directions (from right to left) until you derive the start symbol
- Previous example:
  \[
  \begin{align*}
  id(x) + num(2) * id(y) \\
  & \leftarrow id(x) + num(2) * F \\
  & \leftarrow id(x) + F * F \\
  & \leftarrow id(x) + T * F \\
  & \leftarrow id(x) + T \\
  & \leftarrow F + T \\
  & \leftarrow T + T \\
  & \leftarrow E + T \\
  & \leftarrow E \\
  \end{align*}
  \]
  use \( F ::= id \)
  use \( F ::= num \)
  use \( T ::= F \)
  use \( T ::= T + F \)
  use \( T ::= T * F \)
  use \( F ::= id \)
  use \( F ::= T \)
  use \( E ::= T \)
  use \( E ::= E + T \)

- At each derivation step, need to recognize a handle (the sequence of symbols that matches the right-hand-side of a production)
- Also known as shift-reduce parsing

Predictive Parsing

- The goal is to construct a top-down parser that never backtracks
- Always leftmost derivations
  - left recursion is bad!
- We must transform a grammar in two ways:
  - eliminate left recursion
  - perform left factoring
- These rules eliminate most common causes for backtracking although they do not guarantee a completely backtrack-free parsing

Left Recursion Elimination

- For example, the grammar
  \[
  \begin{align*}
  A & ::= A a \\
  & | b \\
  \end{align*}
  \]
  recognizes the regular expression ba*.
- But a top-down parser may have hard time to decide which rule to use
- Need to get rid of left recursion:
  \[
  \begin{align*}
  A & ::= b A' \\
  A' & ::= a A' \\
  & | \\
  \end{align*}
  \]
  ie, \( A' \) parses the RE a*.
- The second rule is recursive, but not left recursive
Left Recursion Elimination (cont.)

- For each nonterminal X, we partition the productions for X into two groups:
  - one that contains the left recursive productions
  - the other with the rest

That is:

\[
X ::= X a_1 \\
\vdots \\
X ::= X a_n \\
X ::= b_1 \\
\vdots \\
X ::= b_m
\]

where a and b are symbol sequences.

- Then we eliminate the left recursion by rewriting these rules into:

\[
X ::= b_1 X' \\
\vdots \\
X ::= b_m X' \\
X' ::= a_1 X' \\
\vdots \\
X' ::= a_n X' \\
X' ::= \text{id}
\]

Example

\[
E ::= E + T \\
| E - T \\
| T \\
T ::= T * F \\
| T / F \\
| F \\
F ::= \text{id} \\
\]

\[
E' ::= + T E' \\
| - T E' \\
| T \\
T ::= * F T' \\
| / F T' \\
| F T' \\
F ::= \text{id} \\
\]

Left Factoring

- Factors out common prefixes:

\[
X ::= a_1 \\
\vdots \\
X ::= b_n
\]

becomes:

\[
X ::= a X' \\
X' ::= b_1 \\
\vdots \\
X' ::= b_n
\]

Example:

\[
E ::= T + E \\
| T - E \\
| T \\
| \text{id}
\]

Recursive Descent Parsing

\[
E ::= T E' \\
E' ::= + T E' \\
| - T E' \\
| T \\
T ::= T * F T' \\
| T / F T' \\
| F T' \\
F ::= \text{id} \\
\]

\[
\text{static void } E() \{ \text{T(); Eprime(); } \}
\text{static void } Eprime() \{ \\
\quad \text{if (current_token == PLUS)} \\
\quad \{ \text{read_next_token(); T(); Eprime(); } \} \\
\quad \text{else if (current_token == MINUS)} \\
\quad \{ \text{read_next_token(); T(); Eprime(); } \} \\
\quad \} \\
\text{static void } T() \{ \text{F(); Tprime(); } \}
\text{static void } Tprime() \{ \\
\quad \text{if (current_token == TIMES)} \\
\quad \{ \text{read_next_token(); F(); Tprime(); } \} \\
\quad \text{else if (current_token == DIV)} \\
\quad \{ \text{read_next_token(); F(); Tprime(); } \} \\
\quad \} \\
\text{static void } F() \{ \\
\quad \text{if (current_token == NUM || current_token == ID)} \\
\quad \{ \text{read_next_token(); } \} \\
\quad \text{else error(); } \\
\quad \} \\
\]