Parsing  
(Syntax Analysis)

Review: The Front End

Parser
- Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

The Front End

The purpose of the front end is to deal with the input language
- Perform a membership test: code ∈ source language?
- Is the program well-formed (semantically)?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic

The Study of Parsing

The process of discovering a derivation for some sentence
- Need a mathematical model of syntax — a grammar G
- Need an algorithm for testing membership in L(G)
- Need to keep in mind that our goal is building parsers, not studying the mathematics of arbitrary languages

Roadmap
1. Context-free grammars and derivations
2. Top-down parsing  
   → LL(1) parsers; hand-coded recursive descent parsers
3. Bottom-up parsing  
   → Automatically generated LR(1) parsers

Specifying Syntax with a Grammar

Context-free syntax is specified with a context-free grammar

SheepNoise \rightarrow SheepNoise \ pmoo

\pmoo \rightarrow b

This CFG defines the set of noises sheep normally make

It is written in a variant of Backus-Naur form

Formally, a grammar is a four tuple, \( G = (S, \mathcal{N}, \mathcal{T}, \mathcal{P}) \)
- \( S \) is the start symbol
- \( \mathcal{N} \) is a set of non-terminal symbols (syntactic variables)
- \( \mathcal{T} \) is a set of terminal symbols (words)
- \( \mathcal{P} \) is a set of productions or rewrite rules (\( P : N \rightarrow (N \cup \mathcal{T}) \))

Deriving Syntax

We can use the SheepNoise grammar to create sentences

\rightarrow use the productions as rewriting rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SheepNoise</td>
<td>1</td>
<td>SheepNoise</td>
</tr>
<tr>
<td>2</td>
<td>SheepNoise baa</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>:\ baa baa baa</td>
</tr>
</tbody>
</table>

And so on ...
A More Useful Grammar

To explore the uses of CFGs, we need a more complex grammar

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Expr$</td>
<td>$Expr \rightarrow Expr_{op} Expr_{op}$</td>
</tr>
<tr>
<td>$id$</td>
<td>$id$</td>
</tr>
<tr>
<td>$id_{op}$</td>
<td>$id_{op}$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \ast$</td>
<td>$id_{op} \rightarrow \ast$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
</tbody>
</table>

- Such a sequence of rewrites is called a derivation
- Process of discovering a derivation is called parsing

We denote this derivation: $Expr \Rightarrow^* id - num * id$

The Two Derivations for $x - 2 \ast y$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Expr$</td>
<td>$Expr \rightarrow Expr_{op} Expr_{op}$</td>
</tr>
<tr>
<td>$id$</td>
<td>$id$</td>
</tr>
<tr>
<td>$id_{op}$</td>
<td>$id_{op}$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \ast$</td>
<td>$id_{op} \rightarrow \ast$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
<tr>
<td>$id_{op} \rightarrow \div$</td>
<td>$id_{op} \rightarrow \div$</td>
</tr>
</tbody>
</table>

Leftmost derivation

Rightmost derivation

In both cases, $Expr \Rightarrow^* id - num * id$

- The two derivations produce different parse trees
- The parse trees imply different evaluation orders

Derivations and Parse Trees

Another leftmost derivation?

This evaluates as $x - (2 \ast y)$

Derivations and Parse Trees

Two Leftmost Derivations for $x - 2 \ast y$

The Difference:

- Different productions chosen on the second step

Original choice

New choice

Both derivations succeed in producing $x - 2 \ast y$
Ambiguous Grammars

- This grammar allows multiple leftmost derivations for $x \cdot 2 \ast y$
- Hard to automate derivation if $>1$ choice
- The grammar is ambiguous

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$Expr \rightarrow Expr \cdot Expr\ number$</td>
</tr>
<tr>
<td>2</td>
<td>$Expr \rightarrow #$</td>
</tr>
<tr>
<td>3</td>
<td>$Op \rightarrow -$</td>
</tr>
<tr>
<td>4</td>
<td>$Op \rightarrow -$</td>
</tr>
<tr>
<td>5</td>
<td>$id \rightarrow id$</td>
</tr>
<tr>
<td>6</td>
<td>$id \rightarrow id$</td>
</tr>
<tr>
<td>7</td>
<td>$id \rightarrow id$</td>
</tr>
</tbody>
</table>

Derivations and Precedence

These two derivations point out a problem with the grammar:
- It has no notion of precedence, or implied order of evaluation

To add precedence:
- Create a non-terminal for each level of precedence
- Isolate the corresponding part of the grammar
- Force the parser to recognize high precedence subexpressions first

For algebraic expressions
- Multiplication and division, first (level one)
- Subtraction and addition, next (level two)

Note: we are ignoring the issue of associativity for now (remember 3300)

Derivations and Precedence

Adding the standard algebraic precedence produces:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$Goal \rightarrow Expr$</td>
</tr>
<tr>
<td>2</td>
<td>$Expr \rightarrow Expr \cdot Term$</td>
</tr>
<tr>
<td>3</td>
<td>$Term \rightarrow Term \cdot Factor$</td>
</tr>
<tr>
<td>4</td>
<td>$Factor \rightarrow Factor \cdot number$</td>
</tr>
<tr>
<td>5</td>
<td>$Term \rightarrow Term \cdot Factor$</td>
</tr>
<tr>
<td>6</td>
<td>$Factor \rightarrow Factor \cdot number$</td>
</tr>
<tr>
<td>7</td>
<td>$id \rightarrow id$</td>
</tr>
<tr>
<td>8</td>
<td>$Expr \rightarrow Expr \cdot Term$</td>
</tr>
<tr>
<td>9</td>
<td>$Expr \rightarrow Expr \cdot Term$</td>
</tr>
</tbody>
</table>

This grammar is slightly larger
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces some parse tree under leftmost & rightmost derivations
Let’s see how it parses $x \cdot 2 \ast y$

Ambiguous Grammars

Definitions
- If a grammar has more than one leftmost derivation for a single sentential form, the grammar is ambiguous
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is ambiguous
- The leftmost and rightmost derivations for a sentential form may differ, even in an unambiguous grammar

Classic example — the if-then-else problem

$Stmt \rightarrow if\ Expr\ then\ Stmt\ else\ Stmt$

This ambiguity is entirely grammatical in nature

Ambiguity

This sentential form has two derivations

$if\ Expr;\ then\ if\ Expr;\ then\ Stmt;\ else\ Stmt$
**Ambiguity**

Removing the ambiguity
- Must rewrite the grammar to avoid generating the problem
- Match each `else` to innermost unmatched `if`, (common sense rule)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stmt $\rightarrow$ WithElse</td>
</tr>
<tr>
<td>2</td>
<td>NaElse</td>
</tr>
<tr>
<td>3</td>
<td>WithElse $\rightarrow$ <code>if</code> <code>Expr</code> <code>then</code> WithElse <code>else</code> WithElse</td>
</tr>
<tr>
<td>4</td>
<td>OtherStmt</td>
</tr>
<tr>
<td>5</td>
<td>NaElse $\rightarrow$ <code>if</code> <code>Expr</code> <code>then</code> Stmt</td>
</tr>
<tr>
<td>6</td>
<td><code>if</code> <code>Expr</code> <code>then</code> WithElse <code>else</code> NaElse</td>
</tr>
</tbody>
</table>

With this grammar, the example has only one derivation

**Deeper Ambiguity**

Ambiguity usually refers to confusion in the CFG
Overloading can create deeper ambiguity

\[ a = f(17) \]

In many Algol-like languages, `f` could be either a function or a subscripted variable

Disambiguating this one requires context
- Need values of declarations
- Really an issue of `type`, not context-free syntax
- Requires an extra-grammatical solution (not in CFG)
- Must handle these with a different mechanism
  - Step outside grammar rather than use a more complex grammar

**Ambiguity - the Final Word**

Ambiguity arises from two distinct sources
- Confusion in the context-free syntax (if-then-else)
- Confusion that requires context to resolve (overloading)

Resolving ambiguity
- To remove context-free ambiguity, rewrite the grammar
- Change language (e.g., if-then-else)
- To handle context-sensitive ambiguity takes cooperation
  - Knowledge of declarations, types, ...
  - Accept a superset of \( (G) \& \) and check it by other means
  - This is a language design problem

Sometimes, the compiler writer accepts an ambiguous grammar
- Parsing techniques that "do the right thing"
  - `i.e.,` always select the same derivation

**Top-down Parsing**

A top-down parser starts with the root of the parse tree
The root node is labeled with the goal symbol of the grammar

Top-down parsing algorithm:

1. Construct the root node of the parse tree
2. Repeat until the fringe of the parse tree matches the input string
   a. At a node labeled `A`, select a production with `A` on its rhs and, for each symbol on its rhs, construct the appropriate child
   b. When a terminal symbol is added to the fringe and it doesn’t match the fringe, backtrack
3. Find the next node to be expanded (label `NT`)

- The key is picking the right production in step 1
  - That choice should be guided by the input string

**Remember the expression grammar?**

Version with precedence

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal $\rightarrow$ Expr</td>
</tr>
<tr>
<td>2</td>
<td>Expr $\rightarrow$ Expr $*$ Term</td>
</tr>
<tr>
<td>3</td>
<td>Expr $\rightarrow$ Term</td>
</tr>
<tr>
<td>4</td>
<td>Term</td>
</tr>
<tr>
<td>5</td>
<td>Term $\rightarrow$ Term $*$ Factor</td>
</tr>
<tr>
<td>6</td>
<td>Term $\rightarrow$ Factor</td>
</tr>
<tr>
<td>7</td>
<td>Factor</td>
</tr>
<tr>
<td>8</td>
<td>Factor $\rightarrow$ number</td>
</tr>
<tr>
<td>9</td>
<td><code>id</code></td>
</tr>
</tbody>
</table>

And the input `x = 2 * y`
### Example

Let's try $x - 2 \cdot y$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Expr</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>2</td>
<td>Expr + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>3</td>
<td>Term + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>4</td>
<td>Factor + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>5</td>
<td>&lt;id,x&gt; + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>6</td>
<td>&lt;id,x&gt; + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
</tbody>
</table>

*Leftmost derivation, choose productions in an order that exposes problems

### Example

This worked well, except that $-$ doesn't match $-$

The parser must backtrack to here:

### Example

Continuing with $x - 2 \cdot y$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Expr</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>2</td>
<td>Expr + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>3</td>
<td>Term + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>4</td>
<td>Factor + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>5</td>
<td>&lt;id,x&gt; + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>6</td>
<td>&lt;id,x&gt; + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
</tbody>
</table>

This time, "-" and "-" matched

We can advance past "-" to look at $2$:

* Now, we need to expand Term - the last NV on the fringe

### Example

Trying to match the "2" in $x - 2 \cdot y$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>&lt;id,x&gt; - Factor</td>
<td>$x - 12 \cdot y$</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id,x&gt; + num,2</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>8</td>
<td>&lt;id,x&gt; + num,2</td>
<td>$x - 2 \cdot y$</td>
</tr>
</tbody>
</table>

Where are we?

* "2" matches "2"
* We have more input, but no NVs left to expand
* The expansion terminated too soon
* Need to backtrack
Example

Trying again with \(2^2\) in \(x - 2^2\):

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>id, x - Term</td>
<td>(x - 2^2)</td>
</tr>
<tr>
<td>2</td>
<td>id, x - Term</td>
<td>(2^2)</td>
</tr>
<tr>
<td>3</td>
<td>id, x - Factor</td>
<td>(x - 2^2)</td>
</tr>
<tr>
<td>4</td>
<td>id, x - Factor</td>
<td>(2^2)</td>
</tr>
<tr>
<td>5</td>
<td>id, x - Term* Factor</td>
<td>(x - 2^2)</td>
</tr>
<tr>
<td>6</td>
<td>id, x - Term* Factor</td>
<td>(2^2)</td>
</tr>
<tr>
<td>7</td>
<td>id, x -TERM* Factor</td>
<td>(x - 2^2)</td>
</tr>
<tr>
<td>8</td>
<td>id, x - TERM* Factor</td>
<td>(2^2)</td>
</tr>
<tr>
<td>9</td>
<td>id, x -TERM, y</td>
<td>(x - 2^2)</td>
</tr>
<tr>
<td>10</td>
<td>id, x -TERM, y</td>
<td>(2^2)</td>
</tr>
</tbody>
</table>

This time, we matched & consumed all the input

\(\Rightarrow\) Success!

Left Recursion

Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is left-recursive if \(\exists A \in NT\) such that

\(\exists\) a derivation \(A \rightarrow^* Aa\), for some string \(a \in (NT \cup T)\)

Our expression grammar is left-recursive:

\(\cdot\) This can lead to non-termination in a top-down parser

\(\cdot\) For a top-down parser, any recursion must be right recursion

\(\cdot\) We would like to convert the left recursion to right recursion

Non-termination is a bad property in any part of a compiler

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

\[
\text{Fee} \rightarrow \text{Fee} \alpha \\
\mid \beta
\]

where neither \(\alpha\) nor \(\beta\) contain \(\text{Fee}\)

We can rewrite this as

\[
\text{Fee} \rightarrow \beta \text{Fee} \\
\mid \epsilon
\]

where \(\text{Fee}\) is a new non-terminal

This accepts the same language, but uses only right recursion

Eliminating Left Recursion

The expression grammar contains two cases of left recursion

\[
\text{Expr} \rightarrow \text{Expr} + \text{Term} \\
\mid \text{Expr} - \text{Term} \\
\mid \text{Term} \rightarrow \text{Term} * \text{Factor} \\
\mid \text{Term} / \text{Factor} \\
\mid \epsilon
\]

Applying the transformation yields

\[
\text{Expr} \rightarrow \text{Term Expr}^* \\
\mid \text{Term} \rightarrow \text{Factor Term}^* \\
\mid \text{Expr} \rightarrow \text{Term Expr}^* \\
\mid \text{Term} \rightarrow \text{Factor Term}^* \\
\mid \epsilon
\]

These fragments use only right recursion

They retain the original left associativity

Eliminating Left Recursion

Substituting them back into the grammar yields

\[
\begin{align*}
1 & \text{Goal} \rightarrow \text{Expr} \\
2 & \text{Expr} \rightarrow \text{Term Expr}^* \\
3 & \text{Expr}^* \rightarrow \text{Term Expr}^* \\
4 & \text{Term} \rightarrow \text{Factor Term}^* \\
5 & \epsilon \\
6 & \text{Term} \rightarrow \text{Factor Term}^* \\
7 & \text{Term} \rightarrow \text{Factor Term}^* \\
8 & \epsilon \\
9 & \text{Factor} \rightarrow \text{number} \\
10 & \text{Factor} \rightarrow \text{id} \\
11 & \text{Factor} \rightarrow \text{Expr} \\
12 & \epsilon
\end{align*}
\]

\(\cdot\) This grammar is correct, if somewhat non-intuitive.

\(\cdot\) It is left associative, as was the original

\(\cdot\) A top-down parser will terminate using it.

\(\cdot\) A top-down parser may need to backtrack with it.

* General left recursion removal algorithm p.96 EAC
Left Factoring

What if my grammar does not have the LL(1) property?
⇒ Sometimes, we can transform the grammar

The Algorithm

∀ A ∈ NT,
find the longest prefix α that occurs in two
or more right-hand sides of A
if α ∈ e then replace all of the A productions,
A → αZ | γ
with
A → αZ | γ
Z → β1 | β2 | ... | βn
where Z is a new element of NT
Repeat until no common prefixes remain

Left Factoring

(An example)

Consider the following fragment of the expression grammar

Factor → Identifier
| Identifier [ ExprList ]
| Identifier [ ExprList ]
After left factoring, it becomes

Factor → Identifier Arguments
Arguments → [ ExprList ]
| [ ExprList ]
| ε
This form has the same syntax, with the LL(1) property

LL(1) Example - Grammar & Left Factoring

<table>
<thead>
<tr>
<th>Original Grammar</th>
<th>Left Factored Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal → Expr</td>
<td>Goal → Expr</td>
</tr>
<tr>
<td>Expr → Term * Expr</td>
<td>Expr → Term Expr'</td>
</tr>
<tr>
<td>Term → Factor * Term</td>
<td>Term → * Term</td>
</tr>
<tr>
<td>Factor → num</td>
<td>Factor → num</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
</tr>
</tbody>
</table>

Recursive Descent Parsers

• Description
  ⇒ Top-down parser built from a set of mutually-recursive procedures
  ⇒ Each procedure usually implements a nonterminal from the grammar

• Implementation
  ⇒ Backtracking
    • Choose different production if current choice fails
  ⇒ LL(1) Predictive
    • Compare lookahead token to FIRST+ sets to select production
  ⇒ Utility function
    • match(t) {
      if (lookahead == t)
        lookahead = next_token();
      else error();
    }

MCM 430  Lecture 3 37

MCM 430  Lecture 3 38

MCM 430  Lecture 3 39

MCM 430  Lecture 3 40

MCM 430  Lecture 3 41

MCM 430  Lecture 3 42
Recursive Descent LL(1) Parser Implementation

• Terminals
  → Terminals in the input stream appear as token lookahead
  → Can advance lookahead token using: lookahead ← next_token()

• Nonterminals
  → Every NT is associated with a parsing procedure
  → The parsing procedure for A ∈ NT, proc A
    • Responsible for parsing and consuming any string that can be derived from A
    • Choose to replace A with β for production A → β
      - If lookahead ∈ FIRST(β)
      - Error if lookahead token does not match any production

• Parser is invoked by calling proc S for start symbol S

Recursive Descent Parser Example 1

• Given grammar S → xyz | abc
  → FIRST(‘xyz’) = { x }, FIRST(‘abc’) = { a }

• Parser
  parse_S()
  if (lookahead == ‘x’)
    match(‘x’); // S → xyz
  else if (lookahead == ‘a’)
    match(‘a’); // S → abc
  else error();

Recursive Descent Parser Example 2

• Given grammar S → A B | A → x | y B → z
  → FIRST(A) = { x, y }, FIRST(B) = { z }

• Parser
  parse_S()
  if (lookahead == ‘x’)
    match(‘x’); // A → x
  if (lookahead == ‘y’)
    match(‘y’); // A → y
  else error();
  parse_A(); // S → A
  else error();
  parse_B(); // S → B

Recursive Descent Parser Example 3

• Given grammar S → a S b | ε
  → FIRST(‘aSb’) = { a }, FIRST(‘ε’) = Follow(S) = { b, $ }

• Parser
  parse_S()
  if (lookahead == ‘a’)
    match(‘a’); parse_S(); // S → a S b
  else if (lookahead == ‘$’)
    ; // S → ε
  else error();