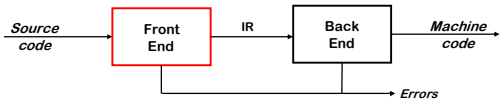
Parsing
(Syntax Analysis)

The Front End

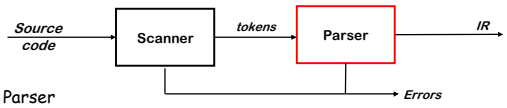


The purpose of the front end is to deal with the input language

- 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

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 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 → *SheepNoise* baa
| baa

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,N,T,P)*

- *S* is the *start symbol* *(set of strings in L(G))*
- *N* is a set of *non-terminal symbols* *(syntactic variables)*
- *T* is a set of *terminal symbols* *(words)*
- *P* is a set of *productions* or *rewrite rules* *($P: N \rightarrow (N \cup T)^+$)*

Deriving Syntax

We can use the *SheepNoise* grammar to create sentences
→ use the productions as *rewriting rules*

Rule	*Sentential Form*
—	*SheepNoise*
2	baa

Rule	*Sentential Form*
—	*SheepNoise*
1	*SheepNoise* baa
2	baa baa

Rule	*Sentential Form*
—	*SheepNoise*
1	*SheepNoise* baa
1	*SheepNoise* baa baa
2	baa baa baa

And so on . . .

A More Useful Grammar

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

1	*Expr*	→	*Expr Op Expr*
2		|	number
3		|	id
4	*Op*	→	+
5		|	-
6		|	*
7		|	/

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
3	‹id,x› *Op Expr*
5	‹id,x› - *Expr*
1	‹id,x› - *Expr Op Expr*
2	‹id,x› - ‹num,2› *Op Expr*
6	‹id,x› - ‹num,2› * *Expr*
3	‹id,x› - ‹num,2› * ‹id,y›

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

We denote this derivation: *Expr* ⇒* id - num * id

Derivations

- At each step, we choose a non-terminal to replace
- Different choices can lead to different derivations

Two derivations are of interest

- *Leftmost derivation* — replace leftmost NT at each step; generates left sentential forms ($\Rightarrow^*_{lm}$)
- *Rightmost derivation* — replace rightmost NT at each step; generates right sentential forms ($\Rightarrow^*_{rm}$)

These are the two *systematic* derivations

(We don't care about randomly-ordered derivations!)

The example on the preceding slide was a *leftmost* derivation

- Of course, there is also a *rightmost* derivation
- Interestingly, the resulting parse trees may be different

The Two Derivations for *x - 2 * y*

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
3	‹id,x› *Op Expr*
5	‹id,x› - *Expr*
1	‹id,x› - *Expr Op Expr*
2	‹id,x› - ‹num,2› *Op Expr*
6	‹id,x› - ‹num,2› * *Expr*
3	‹id,x› - ‹num,2› * ‹id,y›

Leftmost derivation

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
3	*Expr Op* ‹id,y›
6	*Expr* * ‹id,y›
1	*Expr Op Expr* * ‹id,y›
2	*Expr Op* ‹num,2› * ‹id,y›
5	*Expr* - ‹num,2› * ‹id,y›
3	‹id,x› - ‹num,2› * ‹id,y›

Rightmost derivation

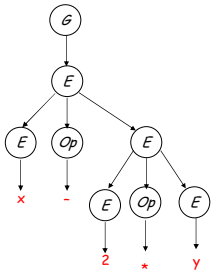
In both cases, *Expr* ⇒* id - num * id

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

Derivations and Parse Trees

Leftmost derivation

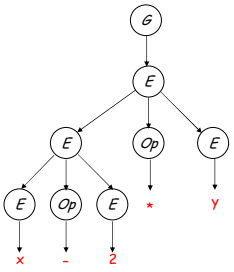
Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
3	‹id,x› *Op Expr*
5	‹id,x› - *Expr*
1	‹id,x› - *Expr Op Expr*
2	‹id,x› - ‹num,2› *Op Expr*
6	‹id,x› - ‹num,2› * *Expr*
3	‹id,x› - ‹num,2› * ‹id,y›



This evaluates as x - (2 * y)

Derivations and Parse Trees

Another leftmost derivation?



This evaluates as (x - 2) * y

Two Leftmost Derivations for *x - 2 * y*

The Difference:

- Different productions chosen on the second step

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
(3)	‹id,x› *Op Expr*
5	‹id,x› - *Expr*
1	‹id,x› - *Expr Op Expr*
2	‹id,x› - ‹num,2› *Op Expr*
6	‹id,x› - ‹num,2› * *Expr*
3	‹id,x› - ‹num,2› * ‹id,y›

Original choice

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
(1)	*Expr Op Expr Op Expr*
3	‹id,x› *Op Expr Op Expr*
5	‹id,x› - *Expr Op Expr*
2	‹id,x› - ‹num,2› *Op Expr*
6	‹id,x› - ‹num,2› * *Expr*
3	‹id,x› - ‹num,2› * ‹id,y›

New choice

- Both derivations succeed in producing *x - 2 * y*

Ambiguous Grammars

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

1	*Expr*	→	*Expr Op Expr*
2		|	number
3		|	id
4	*Op*	→	+
5		|	-
6		|	*
7		|	/

Rule	*Sentential Form*
—	*Expr*
1	*Expr Op Expr*
1	*Expr Op Expr Op Expr*
3	<id,x> *Op Expr Op Expr*
5	<id,x> - *Expr Op Expr*
2	<id,x> - <num,2> *Op Expr*
6	<id,x> - <num,2> * *Expr*
3	<id,x> - <num,2> * <id,y>

different choice than the first time

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 330!)

Derivations and Precedence

Adding the standard algebraic precedence produces:

	1	*Goal*	→	*Expr*
level two	2	*Expr*	→	*Expr + Term*
	3		|	*Expr - Term*
	4		|	*Term*
level one	5	*Term*	→	*Term * Factor*
	6		|	*Term / Factor*
	7		|	*Factor*
	8	*Factor*	→	number
	9		|	id

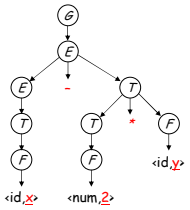
This grammar is slightly larger
- Takes more rewriting to reach some of the terminal symbols
- Encodes expected precedence
- Produces same parse tree under leftmost & rightmost derivations

*Let's see how it parses x - 2 * y*

Derivations and Precedence

Rule	*Sentential Form*
—	*Goal*
1	*Expr*
3	*Expr - Term*
5	*Expr - Term * Factor*
9	*Expr - Term * <id,y>*
7	*Expr - Factor * <id,y>*
8	*Expr - <num,2> * <id,y>*
4	*Term - <num,2> * <id,y>*
7	*Factor - <num,2> * <id,y>*
9	<id,x> - <num,2> * <id,y>

The rightmost derivation



Its parse tree

This produces x - (2 * y), along with an appropriate parse tree.
Both the leftmost and rightmost derivations give the same expression, because the grammar directly encodes the desired precedence.

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 → if *Expr* then *Stmt*
| if *Expr* then *Stmt* else *Stmt*
| *... other stmts ...*

This ambiguity is entirely grammatical in nature

Ambiguity

This sentential form has two derivations

if *Expr*$_1$ then if *Expr*$_2$ then *Stmt*$_1$ else *Stmt*$_2$

Ambiguity

Removing the ambiguity

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

1	*Stmt*	→	*WithElse*
2		|	*NoElse*
3	*WithElse*	→	if *Expr* then *WithElse* else *WithElse*
4		|	*OtherStmt*
5	*NoElse*	→	if *Expr* then *Stmt*
6		|	if *Expr* then *WithElse* else *NoElse*

With this grammar, the example has only one derivation

Ambiguity

if $Expr_1$ then if $Expr_2$ then $Stmt_1$ else $Stmt_2$

Rule	*Sentential Form*
—	*Stmt*
2	*NoElse*
5	if *Expr* then *Stmt*
?	if E_1 then *Stmt*
1	if E_1 then *WithElse*
3	if E_1 then if *Expr* then *WithElse* else *WithElse*
?	if E_1 then if E_2 then *WithElse* else *WithElse*
4	if E_1 then if E_2 then S_1 else *WithElse*
4	if E_1 then if E_2 then S_1 else S_2

This binds the else controlling S_2 to the inner if

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 … endif)
- To handle context-sensitive ambiguity takes cooperation
  - → Knowledge of declarations, types, …
  - → Accept a superset of *L(G)* & 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:

Construct the root node of the parse tree
Repeat until the fringe of the parse tree matches the input string

1. *At a node labeled A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child*
2. *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

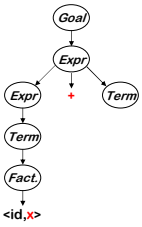
1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id

And the input x - 2 * y

Example

Let's try x – 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
2	*Expr* + *Term*	↑x – 2 * y
4	*Term* + *Term*	↑x – 2 * y
7	*Factor* + *Term*	↑x – 2 * y
9	<id,x> + *Term*	↑x – 2 * y
9	<id,x> + *Term*	x ↑– 2 * y

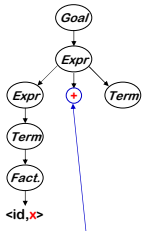


Leftmost derivation, choose productions in an order that exposes problems

Example

Let's try x – 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
2	*Expr* + *Term*	↑x – 2 * y
4	*Term* + *Term*	↑x – 2 * y
7	*Factor* + *Term*	↑x – 2 * y
9	<id,x> + *Term*	↑x – 2 * y
9	<id,x> + *Term*	x ↑– 2 * y



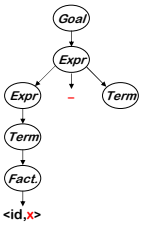
This worked well, except that "–" doesn't match "+"

The parser must backtrack to here

Example

Continuing with x – 2 * y :

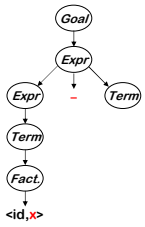
Rule	Sentential Form	Input
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
3	*Expr* – *Term*	↑x – 2 * y
4	*Term* – *Term*	↑x – 2 * y
7	*Factor* – *Term*	↑x – 2 * y
9	<id,x> – *Term*	↑x – 2 * y
9	<id,x> – *Term*	x ↑– 2 * y
—	<id,x> – *Term*	x – ↑2 * y



Example

Continuing with x – 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
3	*Expr* – *Term*	↑x – 2 * y
4	*Term* – *Term*	↑x – 2 * y
7	*Factor* – *Term*	↑x – 2 * y
9	<id,x> – *Term*	↑x – 2 * y
9	<id,x> – *Term*	x ↑– 2 * y
—	<id,x> – *Term*	x – ↑2 * y



This time, "–" and "–" matched

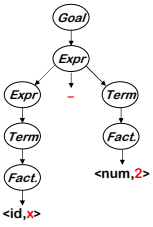
We can advance past "–" to look at "2"

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

Example

Trying to match the "2" in x – 2 * y :

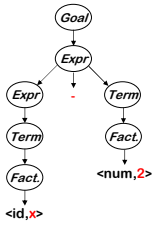
Rule	Sentential Form	Input
—	<id,x> – *Term*	x – ↑2 * y
7	<id,x> – *Factor*	x – ↑2 * y
9	<id,x> – <num,2>	x – ↑2 * y
—	<id,x> – <num,2>	x – 2 ↑* y



Example

Trying to match the "2" in x – 2 * y :

Rule	Sentential Form	Input
—	<id,x> – *Term*	x – ↑2 * y
7	<id,x> – *Factor*	x – ↑2 * y
9	<id,x> – <num,2>	x – ↑2 * y
—	<id,x> – <num,2>	x – 2 ↑* y



Where are we?

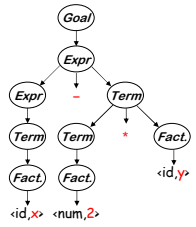
- "2" matches "2"
- We have more input, but no *NT*s left to expand
- The expansion terminated too soon

⇒ Need to backtrack

Example

Trying again with "2" in x - 2 * y :

Rule	Sentential Form	Input
—	<id,x> - *Term*	x - ↑2 * y
5	<id,x> - *Term* * *Factor*	x - ↑2 * y
7	<id,x> - *Factor* * *Factor*	x - ↑2 * y
8	<id,x> - <num,2> * *Factor*	x - ↑2 * y
—	<id,x> - <num,2> * *Factor*	x - 2 ↑* y
—	<id,x> - <num,2> * *Factor*	x - 2 * ↑y
9	<id,x> - <num,2> * <id,y>	x - 2 * ↑y
—	<id,x> - <num,2> * <id,y>	x - 2 * y↑



This time, we matched & consumed all the input

⇒ Success!

Another possible parse

Other choices for expansion are possible

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr* + *Term*	↑x - 2 * y
2	*Expr* + *Term* + *Term*	↑x - 2 * y
2	*Expr* + *Term* + *Term* + *Term*	↑x - 2 * y
2	*Expr* + *Term* + *Term* + …+ *Term*	↑x - 2 * y

consuming no input !

This doesn't terminate *(obviously)*

- Wrong choice of expansion leads to non-termination
- Non-termination is a bad property for a parser to have
- Parser must make the right choice

Left Recursion

Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is *left recursive* if ∃ $A \in NT$ such that
∃ a derivation $A \Rightarrow^+ A\alpha$, for some string $\alpha \in (NT \cup T)^+$

Our expression grammar is left recursive

- This can lead to non-termination in a top-down parser
- For a top-down parser, any recursion must be right recursion
- 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

Fee → *Fee* α
| β

where neither α nor β start with *Fee*

We can rewrite this as

Fee → β *Fie*
Fie → α *Fie*
| ε

where *Fie* 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

Expr → *Expr* + *Term*
| *Expr* - *Term*
| *Term*

Term → *Term* * *Factor*
| *Term* / Factor
| *Factor*

Applying the transformation yields

Expr → *Term Expr'*
Expr' | + *Term Expr'*
| - *Term Expr'*
| ε

Term → *Factor Term'*
Term' | * *Factor Term'*
| / *Factor Term'*
| ε

These fragments use only right recursion

They retain the original left associativity

Eliminating Left Recursion

Substituting them back into the grammar yields

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr'*
3	*Expr'*	→	+ *Term Expr'*
4		|	- *Term Expr'*
5		|	ε
6	*Term*	→	*Factor Term'*
7	*Term'*	→	* *Factor Term'*
8		|	/ *Factor Term'*
9		|	ε
10	*Factor*	→	number
11		|	id
12		|	(*Expr*)

- This grammar is correct, if somewhat non-intuitive.
- It is left associative, as was the original
- A top-down parser will terminate using it.
- 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

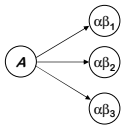
> ∀ *A* ∈ *NT*,
> *find the longest prefix* α *that occurs in two or more right-hand sides of A*
> *if* α ≠ ε *then replace all of the A productions,*
> $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$,
> *with*
> $A \rightarrow \alpha Z \mid \gamma$
> $Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
> *where Z is a new element of NT*
> *Repeat until no common prefixes remain*

Left Factoring

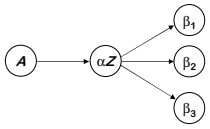
A graphical explanation for the same idea

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$

becomes …



$A \rightarrow \alpha Z$
$Z \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$



Left Factoring (An example)

Consider the following fragment of the expression grammar

Factor	→	Identifier
	|	Identifier [*ExprList*]
	|	Identifier (*ExprList*)

FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { Identifier }
FIRST(rhs_3) = { Identifier }

After left factoring, it becomes

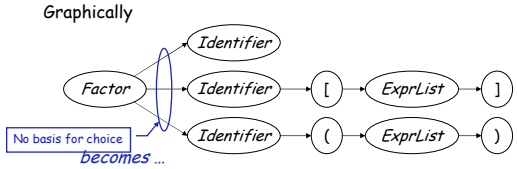
Factor	→	Identifier *Arguments*
Arguments	→	[*ExprList*]
	|	(*ExprList*)
	|	ε

FIRST(rhs_1) = { Identifier }
FIRST(rhs_2) = { [}
FIRST(rhs_3) = { (}
FIRST(rhs_4) = FOLLOW(*Factor*)
⇒ It has the *LL(1)* property

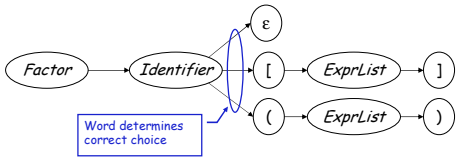
This form has the same syntax, with the *LL(1)* property

Left Factoring

Graphically



becomes …



LL(1) Example – Grammar & Left Factoring

Original Grammar

Goal	→	Expr
Expr	→	Term + Expr
	|	Term – Expr
	|	Term
Term	→	Factor * Term
	|	Factor / Term
	|	Factor
Factor	→	num
	|	id

Left Factored Grammar

Goal	→	Expr
Expr	→	Term Expr'
Expr'	→	+ Expr'
	|	– Expr'
	|	**ε**
Term	→	Factor Term'
Term'	→	* Term
	|	/ Term
	|	**ε**
Factor	→	num
	|	id

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( );
}
```

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 $A \rightarrow \beta$
      - 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"); match("y"); match("z");      // S → xyz
    }
    else if (lookahead == "a") {
      match("a"); match("b"); match("c");      // 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") ||
        (lookahead == "y"))
      parse_A( );      // S → A
    else if (lookahead == "z")
      parse_B( );      // S → B
    else error( );
}
```

```
parse_A( ) {
    if (lookahead == "x")
      match("x");      // A → x
    else if (lookahead == "y")
      match("y");      // A → y
    else error( );
}
parse_B( ) {
    if (lookahead == "z")
      match("z");      // B → z
    else error( );
}
```


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( ); match("b");      // S → a S b
    } else if ((lookahead == "b") || (lookahead == $))
      ;                                        // S → ε
    else error( );
}
```