Parsing

Dolors Sala

Bibliography

- Henri Casanova, Machine learning and systems programming ICS312 (Module 9 Compiling), University of Hawaii (Parsing Lecture)

- Yannis Smaragdakis, Compilers K31, University of Athens, (Lecture 4 Top-down Parsers)

Syntactic Analysis

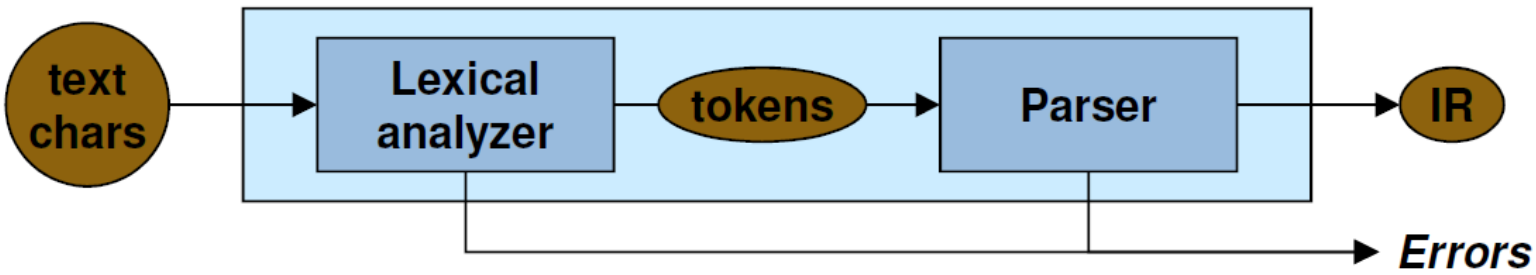
ENGINYERIA UPF

- Lexical analysis is about ensuring that we extract a set of valid words (i.e. tokens/lexemes) from the source code
- But nothing says that the words make a coherent sentence (i.e. program)
- Example:
  - "while A smaller than 3 A + 1 = B"
  - Lexer produces: <TOKEN_WHILE> <TOKEN_NAME A> <TOKEN_NAME smaller> <TOKEN_NAME than> <TOKEN_INTEGER 3> <TOKEN_NAME A> <TOKEN_PLUS> <TOKEN_INTEGER 1> <TOKEN_EQUAL> <TOKEN_NAME B>
  - This line of code is lexically correct but syntactically incorrect

Parsing

- Parsing organizes tokens into sentences

- Important to indicate the errors when there is no match



Grammar

- How do we know a sentence is syntactically correct?
- Check it against a grammar
- A grammar consists of a set of rules that determine which sentences are correct
- Example
  - In English: A sentence must have a verb
  - In C: an “if” goes together with an “else”
- Given a grammar, we can derive sentences by repeated substitution
- Parsing is the reverse process, given a sentence, find a derivation

Specifying Programming Languages

ENGINYERIA UPF

- Regular expressions is a way we have seen for specifying a set of rules
- But they are not enough to describe the syntax of programming languages
- Example: If we have 3 "{" the code has to have 3 "}"
  - It cannot be expressed in a regular expression
  - Regular expressions cannot count and remembering counts
- We need the Context-Free Grammars (CFG)

Context Free Grammars (CFG)

ENGINYERIA UPF

- A context free grammar (CFG) is a set of rules, called production rules
- Each rule describes how a non-terminal symbol can be replaced or expanded by a string that consists of non-terminal and terminal symbols
  - Terminal symbols are tokens (from lexer)
  - Rules are written with a syntax like regular expressions
- Rules can be applied recursively
- Eventually it reaches to an string of only terminal symbols → this sentence is syntactically correct

CFG Specification

ENGINYERIA UPF

- A context free grammar (CFG) is specified with
  - A set of terminal symbols
  - A set of non-terminal symbols
  - A start symbol (S)
  - A set of production rules

CFG Example

ENGINYERIA UPF

- Set of non-terminals: A, B, C (uppercase letters)
- Start non-terminal: S
- Set of terminal symbols: a, b, c, d (lowercase)
- Set of production rules:
  1. S→ A | BC
  2. A → Aa | a
  3. B → bBCd | d
  4. C → dCcd | c
- Doing derivations of the rules we can produce valid strings of this grammar
  - Example: S → BC → bBCdC → bdCdC → bdcdC →bdcdc

CFG Example: A (reduced) grammar for expressions

ENGINYERIA UPF

Production rules:

1. Letter = [a-z]
2. Op = [+-*/] meaning “+” | “-” | “*” | “/”
3. Digit = [0-9]
4. Expr = Expr Op Expr
5. Expr = Number | Identifier
6. Identifier = Letter | Letter identifier
7. Number = Digit Number | Digit

Example: 23+b

Expr → Expr Op Expr → Number Op Expr → Digit Number Op Expr → 2 Digit Op Expr →23 Op Expr → 23 + Expr → 23 + Identifier → 23 + Letter → 23 + b

CFG Example: The grammar specification

1. Set of non terminals ={Expr, Op, Number, Identifier, Letter, Number, Digit}
2. Set of terminals=[a-z0-9+-*/]
3. Start Symbol= Expr
4. Production rules:
    1. Letter = [a-z]
    2. Op = [+-*/]
    3. Digit = [0-9]
    4. Expr = Expr Op Expr
    5. Expr = Number | Identifier
    6. Identifier = Letter | Letter identifier
    7. Number = Digit Number | Digit

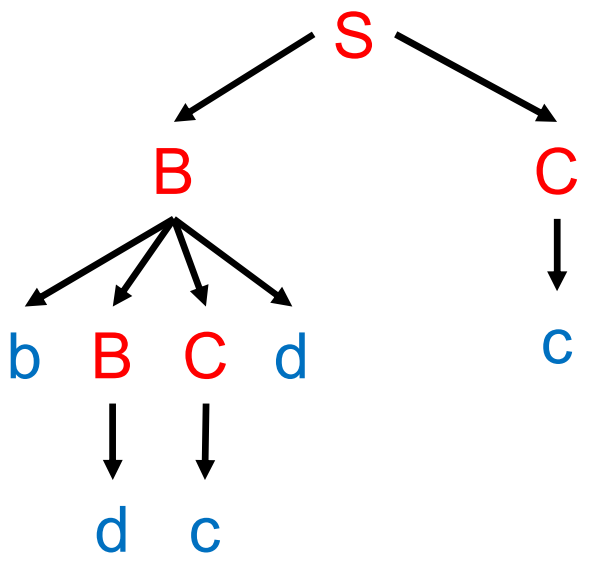
What is Parsing?

ENGINYERIA UPF

- Parsing is the process of discovering a sequence of rule derivations that produce a particular string of non-terminals
- When we say that we cannot parse the string means: we cannot find any combination of rule derivations that start from the start symbol and ends with the string
- A compiler needs a parser: a program that takes in a sequence of tokens (terminal symbols) and discovers a derivation sequence, thus validating that the input is a syntactically correct program

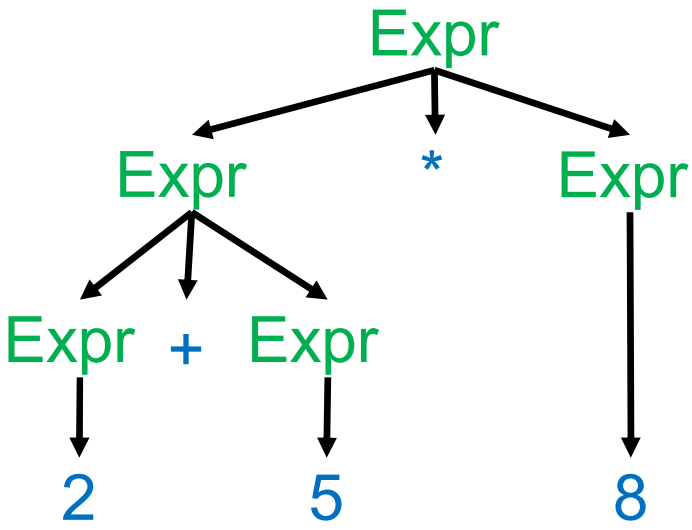
Derivations as Trees

- A convenient way to represent the sequence of derivations is a syntactic tree or parse tree
- Example: S → BC → bBCdC → bdCdC → bdcdC →bdcdc

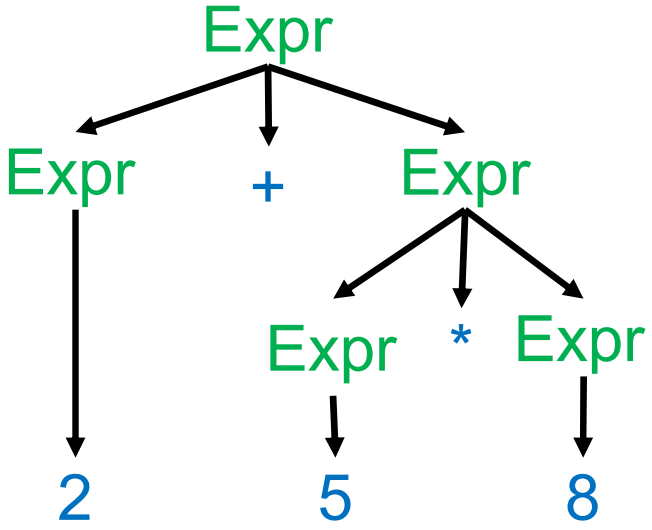


Ambiguity

- We call a grammar ambiguous if a string of terminal symbols can be reached by two different derivation sequences
  - This is the string can have more than one parse tree
- Example: 2+5*8



Left parse-tree



Right parse-tree

Problems with Ambiguity

ENGINYERIA UPF

- Problem: syntax impacts meaning
- We don't know what parse tree will be derived when there are multiple options
- We want unambiguous grammars
- It is possible to modify grammars to make them non-ambiguous
  - By adding non-terminals
  - By adding or rewriting derivation rules
- In the language example: add the rules to consider operator precedence

A non-ambiguous grammar for expressions

1. Letter = [a-z]
2. ~~Op = [+-*/]~~
3. Digit = [0-9]
4. Expr = Term | Expr + Term | Expr – Term
5. Term = Term * Factor | Term / Factor | Factor
6. Factor = Number | Identifier
7. Identifier = Letter | Letter identifier
8. Number = Digit Number | Digit

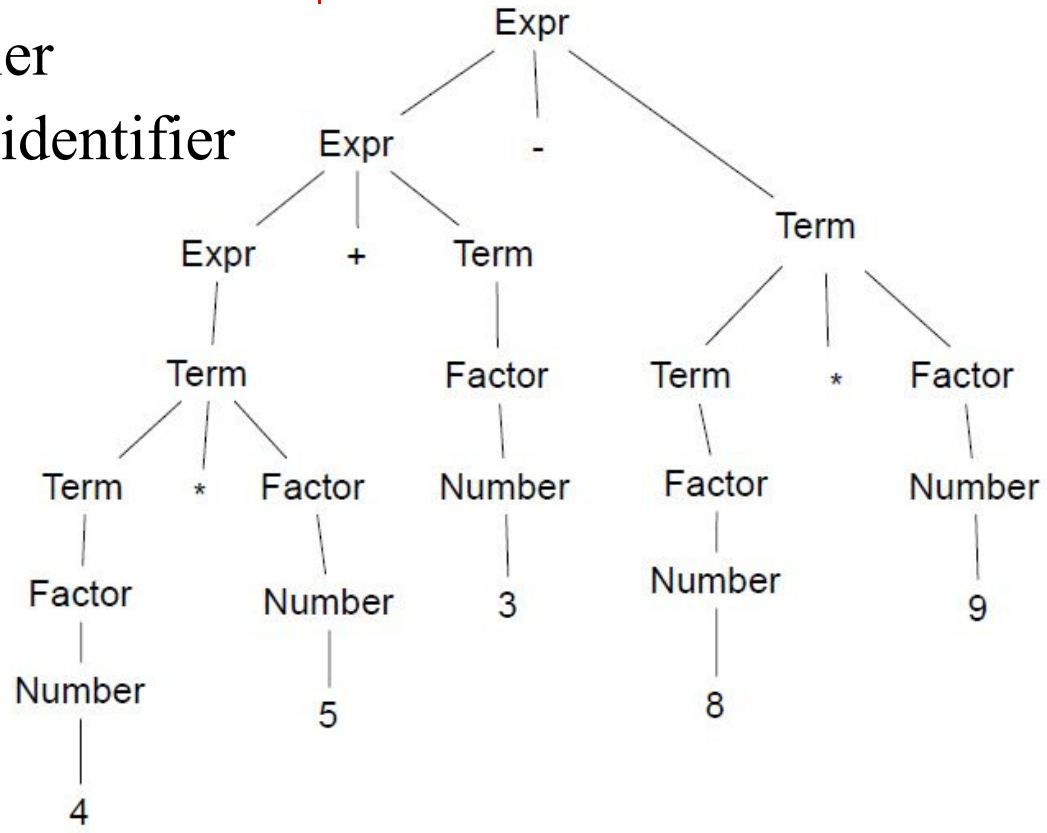
Example: 4*5+3-8*9

1. Letter = [a-z]
2. Op = [+-*/]
3. Digit = [0-9]
4. Expr = Expr Op Expr
5. Expr = Number | Identifier
6. Identifier = Letter | Letter identifier
7. Number = Digit Number | Digit

A non-ambiguous grammar for expressions

1. Letter = [a-z]
2. Digit = [0-9]
3. Expr = Term | Expr + Term | Expr – Term
4. Term = Term * Factor | Term / Factor | Factor
5. Factor = Number | Identifier
6. Identifier = Letter | Letter identifier
7. Number = Digit Number

Example: 4*5+3-8*9



Another Grammar Example: for in C

ENGINYERIA UPF

1. ForStatement = for “(“ StmtCommaList “;” ExprCommaList “;” StmtCommaList “)” “{“ StmtSemicList “}”

2. StmtCommaList = ε | Stmt | Stmt “,” StmtCommaList
3. ExprCommaList = ε | Expr | Expr “,” ExprCommaList
4. StmtSemicList = ε | Stmt | Stmt “;” StmtSemicList

5. Expr = …
6. Stmt = …

```
for(i=0; i < b; i++){
      a+=i;
}
```

A Program Example (sketch)

ENGINYERIA UPF

1. Program = VarDecList FuncDecList
2. VarDecList = ε | VarDecl | VarDecl VarDecList
3. VarDecl = Type IdentCommaList “;”
4. IdentCommaList = Ident | Ident “,” identCommaList
5. Type = int | char | float
6. FuncDeclList = ε | FuncDecl | FuncDecl FuncDeclList
7. FuncDecl = Type Ident “(“ ArgList “)” “{“VardelList StmtList “}”
8. StmtList = ε | Stmt | Stmt StmtList
9. Stmt = Ident “=“ Expr “;” | ForStatement | …
10. Expr = …
11. Ident = …

Real World CFGs

ENGINYERIA UPF

- Some sample grammars
  - LISP 7 rules
  - PROLOG 19 rules
  - Java 30 rules
  - C 60 rules
  - Ada 280 rules
- LISP is particularly easy because
  - No operators, just function calls
  - Therefore no precedence, associativity specified
- LISP is very easy to parse
- In java specification the description of the operator precedence and associativity takes 25 pages

What is parsing?

ENGINYERIA UPF

- Discovering the derivation of a string
  - If one exists
- Harder than generating strings

- Two major approaches
  - Top-down parsing
  - Bottom-up parsing

- Both don't work on all context-free grammars
  - Properties of the grammar determine the parsing approach
  - We may be able to transform a grammar
  - Goal: making parsing efficient

Two Approaches

ENGINYERIA UPF

- Top-down parsers LL(1), recursive descent
  - Start at the root of the parse tree and grow towards leaves
  - Pick a production and try to match the input
  - What happens if the parser chooses the wrong one?

- Bottom-up parsers LR(1), operator precedence
  - Start at the leave and grow toward the root
  - Problem: it may have multiple possible ways to do this
  - Key idea: encode possible parse trees in an internal state (similar to out NFA → DFA conversion)
  - Bottom-up parsers handle a larger class of grammars

Grammars and Parsers

ENGINYERIA UPF

- LL(1) parsers: Top-down
  - Left-to-right input
  - Leftmost derivation
  - 1 symbol of look-ahead

The grammars recognized are LL(1) grammars

- LR(1) parsers: Bottom-up
  - Left-to-right input
  - Rightmost derivation
  - 1 symbol of look-ahead

The grammars recognized are LR(1) grammars

- Others: LL(k), LR(k), …

Top-down Parsing

Example

Grammar (with precedence)

- T= [a-z0-9+-*/]
- NT= {expr, term, factor, number, identifier}
- Start Symbol= expr

Production rules

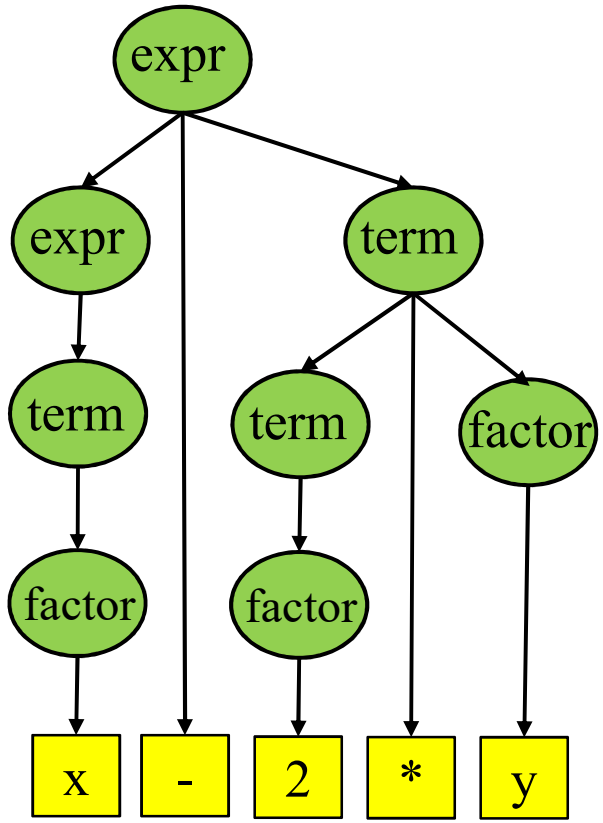
1. expr = expr + term
2. | expr - term
3. | term
4. term = term * factor
5. | expr / term
6. | factor
7. factor = number
8. | identifier

We assume numbers of 1 digit and identifiers of one letter, so we omit the rules of number and identifier (for completeness there should be there, but examples become longer unnecessarily, this way the parsing examples take one step less)

Example: by visual inspection

ENGINYERIA UPF

Input string: x - 2 * y



Rule	Sentential form	Input String
-	expr	↑ x - 2 * y
2	expr - term	↑ x - 2 * y
3	term - term	↑ x - 2 * y
6	factor - term	↑ x - 2 * y
8	id - term	x - ↑2 * y
4	id - term * factor	x - ↑2 * y
6	id - factor * factor	x - ↑2 * y
7	id - number * factor	x - 2 * ↑ y
8	id - number * id	x - 2 * y ↑

↑ current position

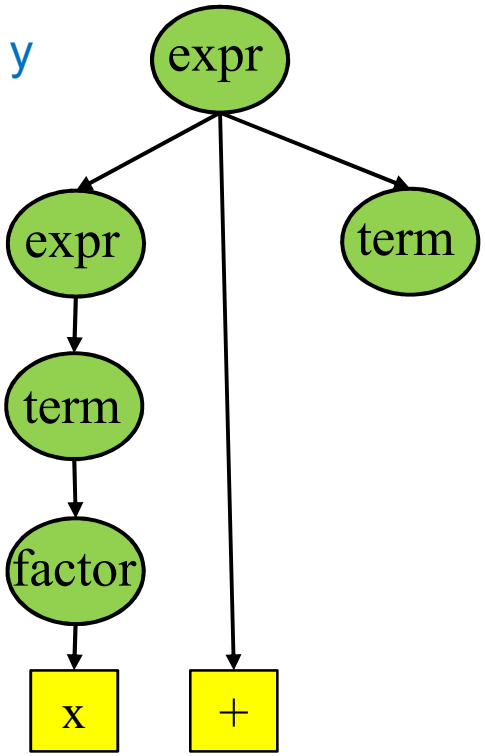
But how to do an automatic process to obtain the correct derivation?

Production rules

1. expr = expr + term
2. | expr - term
3. | term
4. term = term * factor
5. | term / factor
6. | factor
7. factor = number
8. | identifier

Automatic Process: use rules in order

Input string: x - 2 * y



Rule	Sentential form	Input String
-	expr	↑ x - 2 * y
1	expr + term	↑ x - 2 * y
3	term + term	↑ x - 2 * y
6	factor + term	↑ x - 2 * y
8	id + term	x ↑ - 2 * y

fails

↑ current position

Problem:

- We cannot continue
- We guessed wrong at step 2
- What should we do now?
  - Go back to step 2 and take another derivation

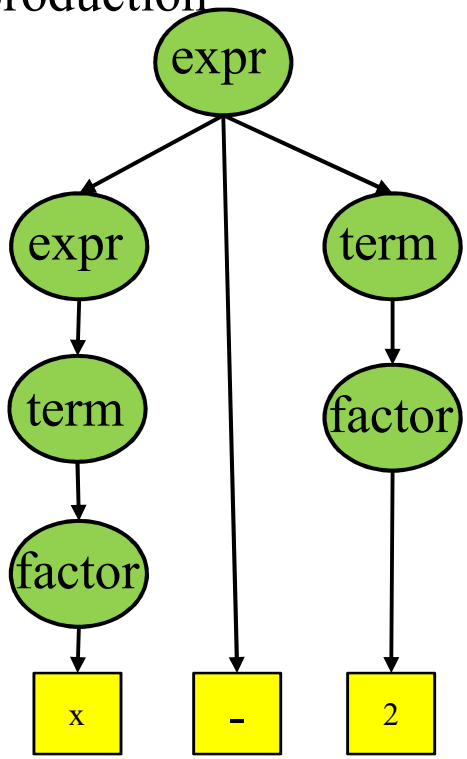
Production rules

1. expr → expr + term
2. | expr - term
3. | term
4. term → term * factor
5. | term / factor
6. | factor
7. factor → number
8. | identifier

Backtracking

If we cannot match the current input position

- Undo the productions
- Choose another production
- Continue



Input string: x - 2 * y

Rule	Sentential form	Input String
-	expr	↑ x - 2 * y
1	expr - term	↑ x - 2 * y
3	term - term	↑ x - 2 * y
6	factor - term	↑ x - 2 * y
8	id - term	x - ↑ 2 * y
6	id - factor	x - ↑ 2 * y
7	id - number	x - 2 ↑ * y

↑ current position

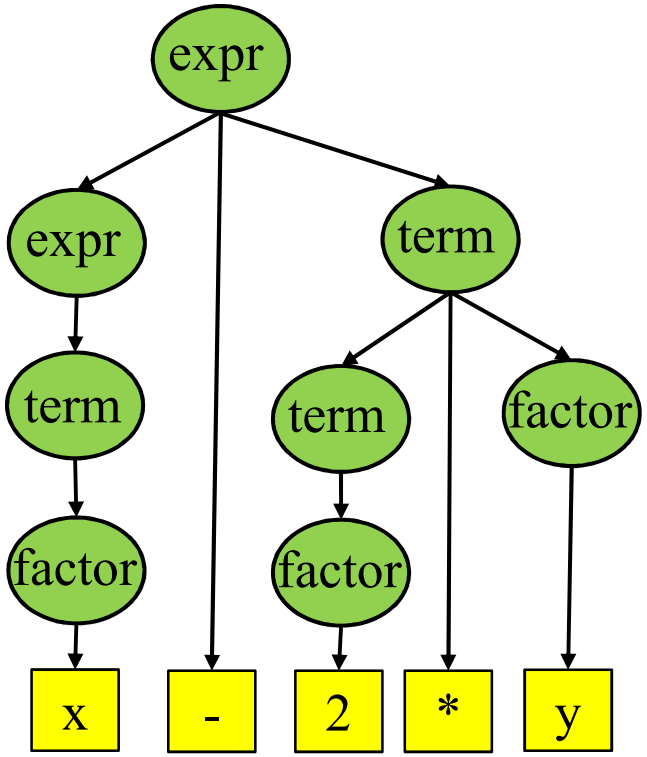
Production rules

1. expr → expr + term
2. | expr - term
3. | term
4. term → term * factor
5. | term / factor
6. | factor
7. factor → number
8. | identifier

Problem:

- More input to read, but no non-terminal to process
- Fail: backtrack again

Successful Parse



Input string: x - 2 * y

Rule	Sentential form	Input String
-	expr	↑ x - 2 * y
1	expr - term	↑ x - 2 * y
3	term - term	↑ x - 2 * y
6	factor - term	↑ x - 2 * y
8	id - term	x - ↑ 2 * y
4	id – term * factor	x - ↑ 2 * y
6	id – factor * factor	x - ↑ 2 * y
7	id – num * factor	x - 2 * ↑ y
8	id – num * id	x - 2 * y ↑

↑ current position

- The final derivation tree has all non-terminals in the leaves, when finish to process all input string

- The goal of a parser: an automatic process that tries all the possibilities and finds the derivation tree if exists

Top-down Parsing

ENGINYERIA UPF

- Start with the root of the parse tree
  - Root of the tree: node labeled with the start symbol

Algorithm

1. Repeat until the parse tree matches the input string
   1. At node A, select one of A's possible productions
      1. Add a child node for each symbol on (rhs) derivation rule
   2. Find the next (non-terminal) node to expand

2. Done when:
   1. Leaves of parse tree match the input string (success)
   2. No more derivations to apply but don't match (fail)

Infinite Expansion Problem

ENGINYERIA UPF

- Rules that do not consume any input value, but expand the tree with additional non-terminal factors, can produce an infinite substitution
  - An automatic process would not end

Production rules

1. expr → expr + term
2. | expr - term
3. | term
4. term → term * factor
5. | term / factor
6. | factor
7. factor → number
8. | identifier

Input string: x – 2 * y ↑ current position

Rule	Sentential form	Input String
-	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
..	..	..

- Wrong choice of rule sequence can lead to a non-ending program
  - It may not be as obvious as this
- This grammar is left recursive

Left Recursion

- Formally: A grammar is left recursive if ∃ a non-terminal A such that A→* A α (for some set of symbols α)
- What does →* mean?
  - With one or more derivations it can get again the same symbol in the left side of the derivation (recursive)
  - A → Bx
  - B → Ay
  - After applying these two derivations to A it results to Ayx
    - It is a recursion because it appears the same symbol
    - It is left-side recursive because the same symbol remains (first) in the left side of the expression
- Top-down parsers cannot handle left recursion
- But, left recursion can be removed systematically (automatically)

Notation

- Non-terminals – Capital letter (A, B, C..)
- Terminals – Lowercase underlined letters (x, y, c..)
- A mixed of terminals and non-terminals – Greek letters (α, β, γ …)
- Example

Rule	Production Rule
1	A → B + x
1	A → B α

α = + x

Eliminating left-recursion

ENGINYERIA UPF

- Example

Rule	Production Rule
1	foo → foo α
2	| β

Language?

β followed by 0 or more α's (βα*)

- Rewrite as

Rule	Production Rule
1	foo → β bar
2	bar → α bar
3	| ε

New non-terminal bar
that appears at the right side of the expression

← This production give one β

← These 2 productions give 0 or more α's

Same Language: β followed by 0 or more α's

The new bar non-terminal symbol is on the right, so now the process ends as all rule substitutions imply an advance in the input string (or no expansion of non-terminal symbols)

Language example: Eliminating left-recursion

ENGINYERIA UPF

- Left-recursion rules:
  - 1, 2, 3
  - 4, 5, 6

Rule	Production Rule
1	foo → foo α
2	| β

Rule	Production Rule
1	foo → β bar
2	bar → α bar
3	| ε

- Eliminating left-recursion:

Production rules

1. expr → expr + term
2. | expr - term
3. | term
4. term → term * factor
5. | term / factor
6. | factor
7. factor → number
8. | identifier

1. expr → term expr2
2. expr2 → + term expr2
3. | - term expr2
4. | ε

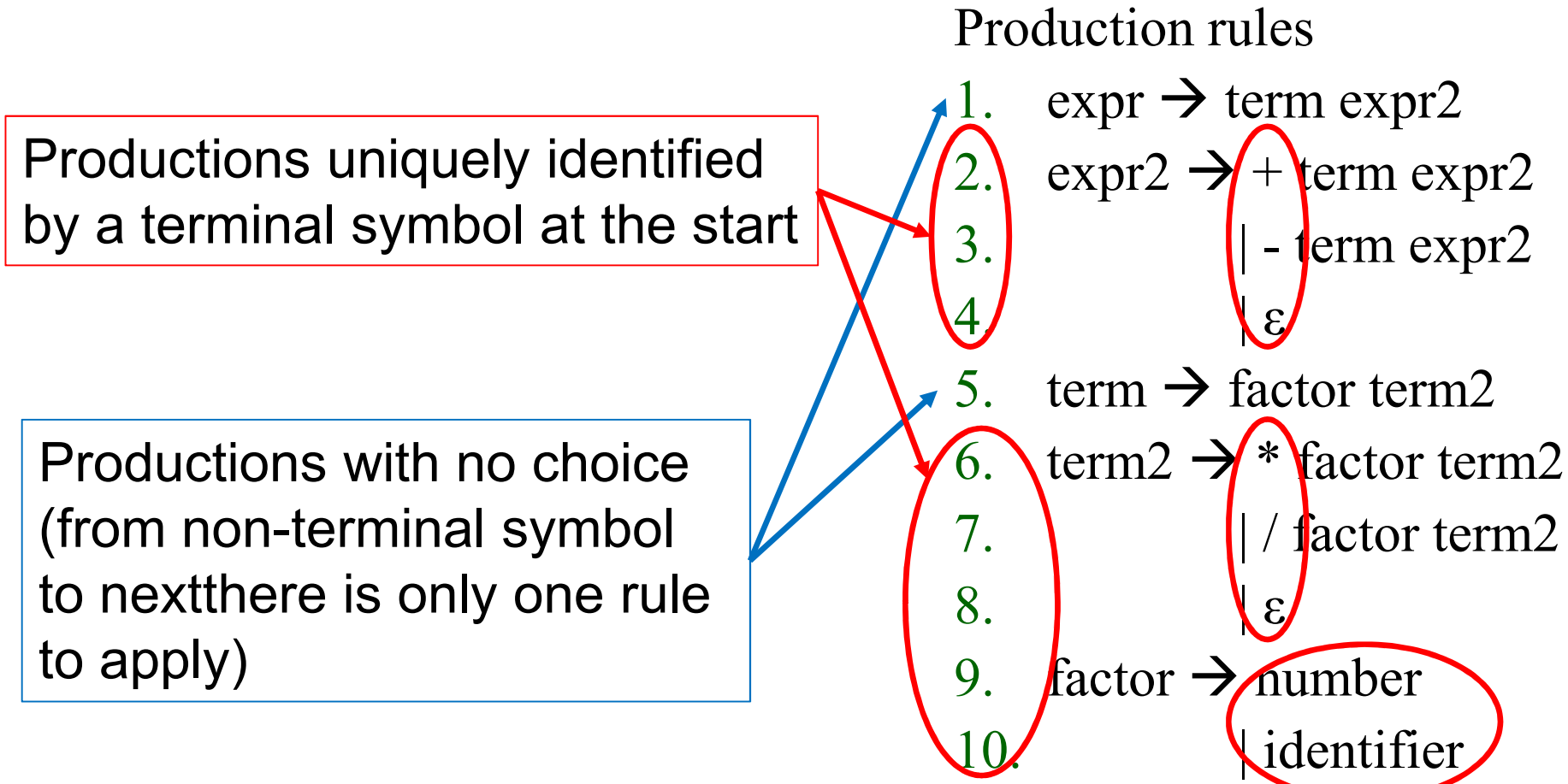
4. term → factor term2
5. term2 → * factor term2
6. | / factor term2
7. | ε

Eliminating left-recursion

ENGINYERIA UPF

- Resulting grammar
  - All right recursive
  - Keeps same definition of the language
  - But not as intuitive to read
- Top-down parser
  - Always terminates
  - Still does backtracking
- There is an algorithm to do it automatically, but we do not do it in this course

Right-recursive grammar



- We can choose the correct production by looking at the next input symbol
  - This is called lookahead

Lookahead

- Goal: avoid backtracking
  - Look at future input symbols
  - Use extra context to make right decision
- How much lookahead is needed?
  - In general, an arbitrary amount for the full class of CFGs
  - Use fancy-dancy algorithm - CYK algorithm $O(n^3)$
- Fortunately
  - Many CFGs can be parsed with limited lookahead
  - Covers most programming languages (not C++, nor Perl)
- We only cover lookahead of 1

Top-down Parsing

ENGINYERIA UPF

- Goal: Given productions A → α | β, the parser should be able choose between α and β
- Trying to match A
  - How can the next input token help us decide?
- Solution: FIRST sets *(almost a solution)*
  - Informally: FIRST(α) is the set of tokens that could appear as the first symbol in a string derived from α
  - Definition: x in FIRST(α) iff α →* xγ
- Building FIRST sets
  - We'll look at the algorithm later

Top-down Parsing

- The LL(1) property
- Given A → α and A → β we would like:
  - FIRST(α) ∩ FIRST(β) = ∅
  - We write FIRST(A = α) as FIRST(α)
- Parser can make right choice by 1 lookahead token
- Almost…what it cannot handle?
  - ε productions

Top-down Parsing : ε productions

- ε productions complicate the definition of LL(1)
- Consider A → α and A→ β and α may be empty
- In this case there is no symbol to identify α
- Example
  - What is FIRST(#4)?

    ={ ε }
  - When we match with production #4?
    - If A is empty what will the next symbol be?
    - It must be one of the symbols that immediately follow an A: z

Rule	Production Rule
1	S → A z
2	A → x B
3	| y C
4	| ε

- Solution: the FOLLOW set

FOLLOW sets

- Build a FOLLOW set for each symbol that could produce ε
- Extra condition for LL:

Rule	Production Rule
1	S → A z
2	A → x B
3	| y C
4	| ε

- Example:
  - FIRST(#2) = { x }
  - FIRST(#3) = { y }
  - FIRST(#4) = { ε }
  - FOLLOW(A) = { z }
  - Now we can uniquely identify each production:
    - If we are trying to match an A and the next token is z, then match production 4

FIRST and FOLLOW sets: Formal Definition

FIRST(α)

Right-hand side symbols

For some $\alpha \in (T \cup NT)^*$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

and $\varepsilon \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \varepsilon$

FOLLOW(A)

For some $A \in NT$, define FOLLOW(A) as the set of symbols that can occur immediately after A in a valid sentence.

FOLLOW(G) = {EOF}, where G is the start symbol

FIRST and FOLLOW sets – Details

ENGINYERIA UPF

- Note
  - FIRST and FOLLOW are sets
  - FIRST may contain ε in addition to other symbols
- Example:
  - FIRST(#2) ?
    - FIRST(B) = { x, y, ε }
  - When do we care about FOLLOW(B)?
    - If FIRST(B) contains ε

Rule	Production Rule
1	S → A z
2	A → B k
4	B → x
5	| y
6	| ε

- Definition: FIRST+(A→ α) as:
  - FIRST(α) ∪ FOLLOW(A) if ε ∈ FIRST(α)
  - FIRST(α), otherwise
- FIRST+(B) = {x, y} ∪ FOLLOW(B) = {x, y, k}

LL(1) Property

- Key idea
  - Build a parse tree top-down
  - Use lookahead token to pick next production
  - Each production must be uniquely identified by the terminal symbols that may appear at the start of strings derived from it
- Definition:

  A grammar is LL(1) iff $A = \alpha$ and $A \rightarrow \beta$ and $\text{FIRST}^+(A \rightarrow \alpha) \cap \text{FIRST}^+(A \rightarrow \beta) = \emptyset$

  (intuitively: when we can decide what production rule to apply next just looking at the next symbol of the input)

Parsing LL(1) grammar

ENGINYERIA UPF

Code: simple, fast routine to recognize each production

Given A→ $\beta_1 \mid \beta_2 \mid \beta_3$ with

$$\text{FIRST}^+(\beta_i) \cap \text{FIRST}^+(\beta_j) = \emptyset \text{ for all } i \mathrel{!}= j$$

```
/* find rule for A */
if (current token ∈ FIRST+(β1))
   select A → β1
else if (current token ∈ FIRST+(β2))
   select A → β2
else if (current token ∈ FIRST+(β3))
   select A → β3
else
   report an error and return false
```

Computing First sets

- Idea: Use FIRST sets of the right-hand side (rhs) of the production $A \rightarrow B_1\ B_2\ B_3\ \ldots$
- Cases
  - $\text{FIRST}(A \rightarrow B) = \text{FIRST}(B_1)$
    - What $\text{FIRST}(B_1)$ mean?
    - Union of $\text{FIRST}(B_1 \rightarrow \gamma)$ for all γ
  - What if ε in $\text{FIRST}(B_1)$?
    - $\text{FIRST}(A \rightarrow B) \cup= \text{FIRST}(B_2)$
  - What if ε in $\text{FIRST}(B_i)$ for all i?
    - $\text{FIRST}(A \rightarrow B) \cup= \{\varepsilon\}$

$\cup$ = *equivalent* +=

Algorithm

- For one production p = A→β

```
if (β is a terminal t)
        FIRST(p) = {t}
else if (β == ε)
        FIRST(p) = {ε}
else
        Given   β = B1 B2 B3 … Bk
        i = 0
        do {    i = i + 1;
                FIRST(p) += FIRST(Bi)  -  {ε}
        } while ( ε  in FIRST(Bi)   &&   i < k)

        if (ε  in FIRST(Bi) && i == k)      FIRST(p) += {ε}
```

1. expr → term expr2
2. expr2 → + term expr2
3. | - term expr2
4. | ε
5. term → factor term2
6. term2 → * factor term2
7. | / factor term2
8. | ε
9. factor → number
10. | identifier

Computing FOLLOW sets

- Idea: Push FOLLOW sets down, use FIRST where needed

$$A \rightarrow B_1\ B_2\ B_3\ B_4\ \ldots\ B_k$$

- Cases:
  - What is FOLLOW(B_1)?
    - FOLLOW(B_1) = FIRST(B_2)
    - In general: FOLLOW(B_i) = FIRST(B_{i+1})
  - What is FOLLOW(B_k)?
    - FOLLOW(B_k) = FOLLOW(A)
  - What if $\varepsilon \in$ FIRST(B_k)?
    - FOLLOW(B_{k-1}) $\cup$= FOLLOW(A)

Example

Production rules

1. goal → expr
2. expr → term expr2
3. expr2 → + term expr2
4. | - term expr2
5. | ε
6. term → factor term2
7. term2 → * factor term2
8. | / factor term2
9. | ε
10. factor → number
11. | identifier

- FIRST(3) = {+}
- FIRST(4) = {-}
- FIRST(5) = {ε}
- FIRST(7) = {*}
- FIRST(8) = {/}
- FIRST(9) = {ε}
- FIRST(1) = ?
- FIRST(1) = FIRST(2)
  = FIRST(6)
  = FIRST(10) ∪ FIRST(11)
  = { number, identifier}

Example

Production rules

1. goal → expr
2. expr → term expr2
3. expr2 → + term expr2
4. | - term expr2
5. | ε
6. term → factor term2
7. term2 → * factor term2
8. | / factor term2
9. | ε
10. factor → number
11. | identifier

- FOLLOW(goal) = {EOF}
- FOLLOW(expr) = FOLLOW(goal) = {EOF}
- FOLLOW(expr2) = FOLLOW(expr) = {EOF}

- FOLLOW(term) = ?
- FOLLOW(term) = FIRST(expr2)
  = {+, -, ε}
  = {+, -, FOLLOW(expr)}
  = {+, -, EOF}
- FOLLOW(term2) = FOLLOW(term)
- FOLLOW(factor) = ?
- FOLLOW(factor) = FIRST(term2)
  = {*, /, ε}
  = {*, /, FOLLOW(term)}
  = {*, /, +, -, EOF}

LL(1) Property

- Recall: a grammar is LL(1) iff

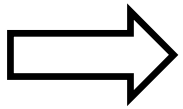
  $A \rightarrow \alpha$ and $A \rightarrow \beta$ and

  $\text{FIRST}+(A \rightarrow \alpha) \cap \text{FIRST}+(A \rightarrow \beta) = \emptyset$

- What if my grammar is not LL(1)?
  - We may be able to fit it with transformations
- Example:

Production rules

1. $A \rightarrow \underline{\alpha}\ \beta_1$
2. $\quad | \underline{\alpha}\ \beta_2$
3. $\quad | \underline{\alpha}\ \beta_3$



Production rules

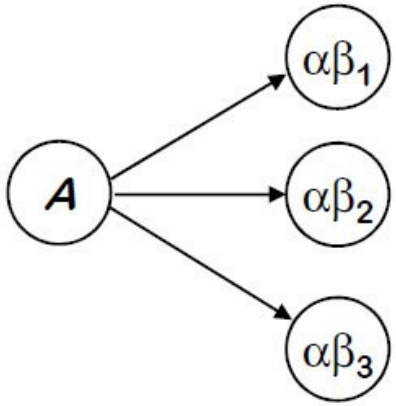
1. $A \rightarrow \underline{\alpha}\ Z$
2. $Z \rightarrow \beta_1$
3. $\quad | \beta_2$
4. $\quad | \beta_3$

Left Factoring

- Graphically

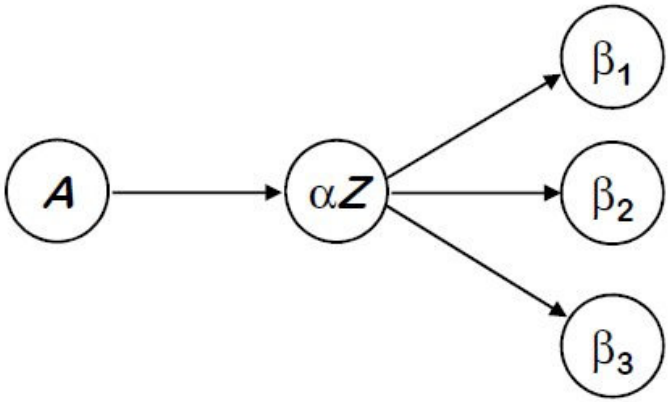
Production rules

1. $A \rightarrow \underline{\alpha}\ \beta_1$
2. $\quad | \underline{\alpha}\ \beta_2$
3. $\quad | \underline{\alpha}\ \beta_3$



Production rules

1. $A \rightarrow \underline{\alpha}\ Z$
2. $Z \rightarrow \beta_1$
3. $\quad | \beta_2$
4. $\quad | \beta_3$



Expression Example

ENGINYERIA UPF

Production rules

1. factor → identifier
2. | identifier [expr]
3. | identifier (expr)

FIRST+(1) = {identifier}
FIRST+(2) = {identifier}
FIRST+(3) = {identifier}

After factoring

Production rules

1. factor → identifier post
2. post → [expr]
3. | (expr)
4. | ε

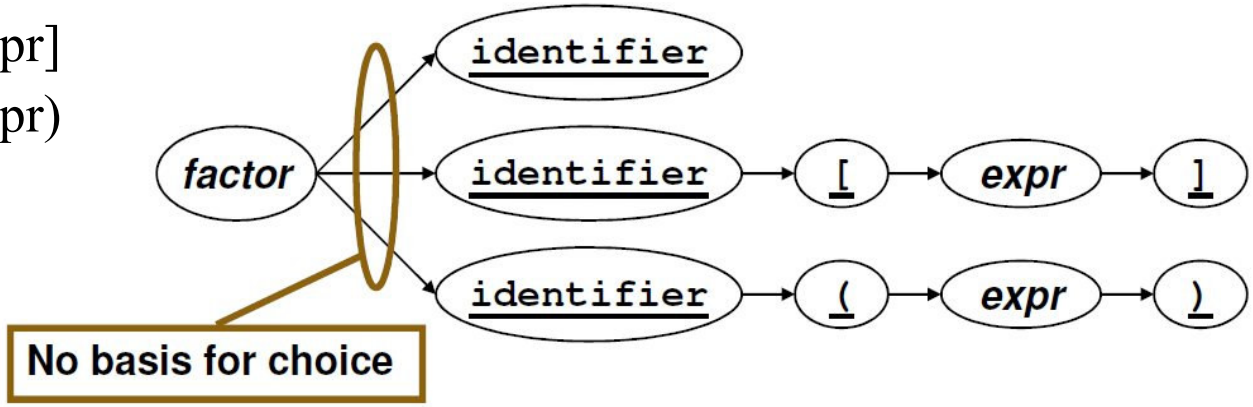
FIRST+(1) = {identifier}
FIRST+(2) = { [}
FIRST+(3) = { (}
FIRST+(4) = ?
= FOLLOW(post)
= { …}

Now it has LL(1) property

Expression Example (Graphically)

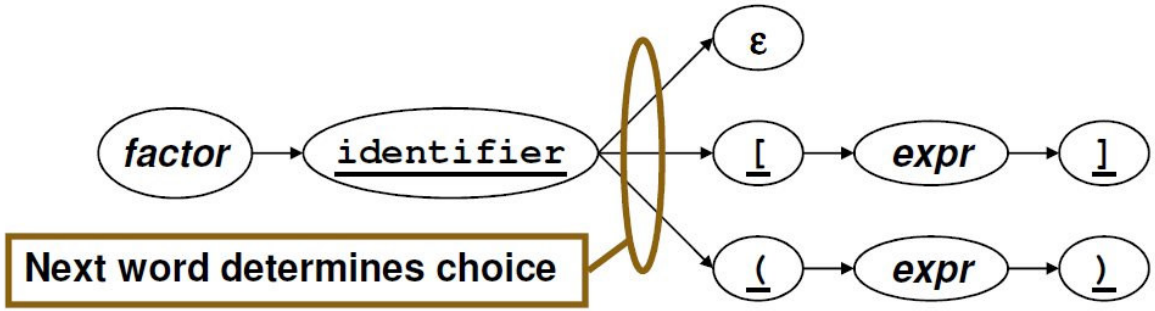
Production rules

1. factor → identifier
2. | identifier [expr]
3. | identifier (expr)



Production rules

1. factor → identifier post
2. post → [expr]
3. | (expr)
4. | ε



Left factoring Reach

- Using left factoring and left recursion, can we turn an arbitrary CFG to a form that it meets the LL(1) property?

- Given a CFG that does not meet LL(1) condition, it is undecidable whether or not an LL(1) grammar exists

Limits of LL(1)

- Example:

$$\{a^n 0 b^n \mid n>=1\} \cup \{a^n 1 b^{2n} \mid n>=1\}$$

no LL(1) grammar

aa0bb

aa1bbbb

Production rules

1. G → a A b
2. | a B bb
3. A → a A b
4. | 0
5. B → a B bb
6. | 1

Problem: it needs an unbounded number of a's before you can determine whether you are in the A group or B group → Not an LL(1) grammar

Predictive Parsing Implementation

ENGINYERIA UPF

- The parser can "predict" the correct expansion
- Using lookahead and FIRST and FOLLOW sets

- Two kinds of predictive parsers implementations
  - Recursive descent
    - Often hand-written
  - Table driven
    - Generate tables from FIRST and FOLLOW sets

Recursive Descent

Production rules

1. goal → expr
2. expr → term expr2
3. expr2 → + term expr2
4. | - term expr2
5. | ε
6. term → factor term2
7. term2 → * factor term2
8. | / factor term2
9. | ε
10. factor → number
11. | identifier

- This produces a parser with six mutually recursive routines:
  - goal
  - expr
  - expr2
  - term
  - term2
  - factor
- Each one recognizes one NT or T
- The term descent refers to the direction in which the parse tree is built

Example Code

- goal Symbol:

```
main()
  /* Match goal --> expr */
  tok = nextToken();
  if (expr() && tok == EOF)
    then proceed to next step;
    else return false;
```

- Top-level expression (expr)

```
expr()
  /* Match expr --> term expr2 */
  if (term() && expr2());
    return true;
  else return false;
```

Note: Eventually this should be generalized to an array of rules, each with rhs and list of lhs terms (to have a code general for any set of rules)

Example Code

- Match expr2:

```
expr2()
  /* Match expr2 --> + term expr2 */
  /* Match expr2 --> - term expr2 */

  if (tok == '+' or tok == '-')
    tok = nextToken();
    if (term())
      then if (expr2())
             return true;
      else return false;

  /* Match expr2 --> empty */
  return true;
```

Note: To meet our requirements the terminal terms should be defined as a CONSTANT definition. Example: SUM_OP = '+'

Top-down parsing

ENGINYERIA UPF

- So far:
  - It gives us a yes or no answer
  - But we want to build the parse tree
  - How?
- Add actions to matching routines
  - Build a new node of the tree (lhs) at each matching rule connecting to all its rhs terms (stored in a stack)

Table-driven Approach

- Encode mapping in a table
  - Row for each non-terminal
  - Column for each terminal symbol

    Table[NT, symbol] = rule#

    If symbol ∈ FIRST+(NT→rhs(#))

Production rules

1. goal → expr
2. expr → term expr2
3. expr2 → + term expr2
4. | - term expr2
5. | ε
6. term → factor term2
7. term2 → * factor term2
8. | / factor term2
9. | ε
10. factor → number
11. | identifier

	+	-	*	/	id	num
expr2	term expr2 r3	term expr2 r4	error	error	error	error
term2	ε, r9	ε, r9	factor term2 r7	factor term2 r8	error	error
factor	error	error	error	error	r11 (do nothing)	r10 (do nothing)

Only shown rules for NT with different options (but all NT raws should be included)

Table-driven Approach

ENGINYERIA UPF

	+	-	*	/	id	num	EOF
goal	error				expr r1		error
expr	error				term expr2 r2		error
expr2	term expr2 r3	term expr2 r4	error				ε r5
term	error				factor term2 r6		error
term2	ε r9		factor term2 r7	factor term2 r8	error		error
factor	error				(nothing) r11	(nothing) r10	error

Summary

- Top-down parsing starts with the start symbol and keeps advancing applying production rules substituting the lhs by the rhs

- To decide what production rule to use, it defines the FIRST, FOLLOW, and FIRST+ sets, for each production rule

- An LL(1) language can decide what production rule to apply by just looking at next input symbol (no conflict)

- Different ways to implement it
  - Recursive Descent: apply a routine for each symbol (the rule)
  - Table Driven: build the table of NT symbols and rules
  - Both use a stack to keep track of the tree → build parse tree