COMPILER DESIGN

Syntactic analysis: Part I

Parsing, derivations, grammar transformation, predictive parsing, introduction to first and follow sets

Syntactical analysis

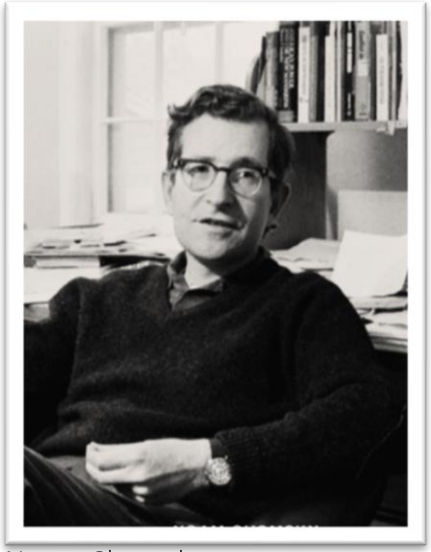
- Syntax analysis involves **parsing** the token sequence to identify the syntactic structure of the program.
- The parser's output is some form of **intermediate representation** of the program's structure, typically a **parse tree**, which replaces the linear sequence of tokens with a tree structure built according to the rules of a **formal grammar** which is used to define the language's syntax.
- This is usually done using a **context-free grammar** which recursively defines components that can make up an valid program and the order in which they must appear.
- The resulting parse tree is then analyzed, augmented, and transformed by later phases in the compiler.
- Parsers are written by hand or generated by parser generators, such as *Yacc*, *Bison*, *ANTLR* or *JavaCC*, among other tools.

Syntactic analyzer

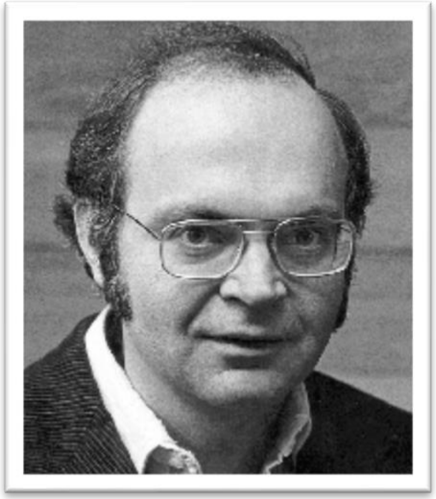
- Roles
  - Analyze the structure of the program and its component **declarations**, **definitions**, **statements** and **expressions**
  - Check for (and recover from) syntax errors
  - Drive the front-end’s execution

Syntax analysis: history

- Historically based on formal natural language grammatical analysis (Chomsky, 1950s).
- Use of a ***generative grammar***:
  - builds sentences in a series of steps;
  - starts from abstract concepts defined by a set of ***grammatical rules*** (often called ***productions***);
  - refines the analysis down to lexical elements.
- Analyzing (parsing) consists in constructing the way in which the sentences can be constructed by the productions.
- Valid sentences can be represented as a ***parse tree.***
- Constructs a ***proof***, called a ***derivation***, that the grammatical rules of the language can generate the sequence of tokens given in input.
- Most of the standard parsing algorithms were invented in the 1960s.
- Donald Knuth is often credited for clearly expressing and popularizing them.



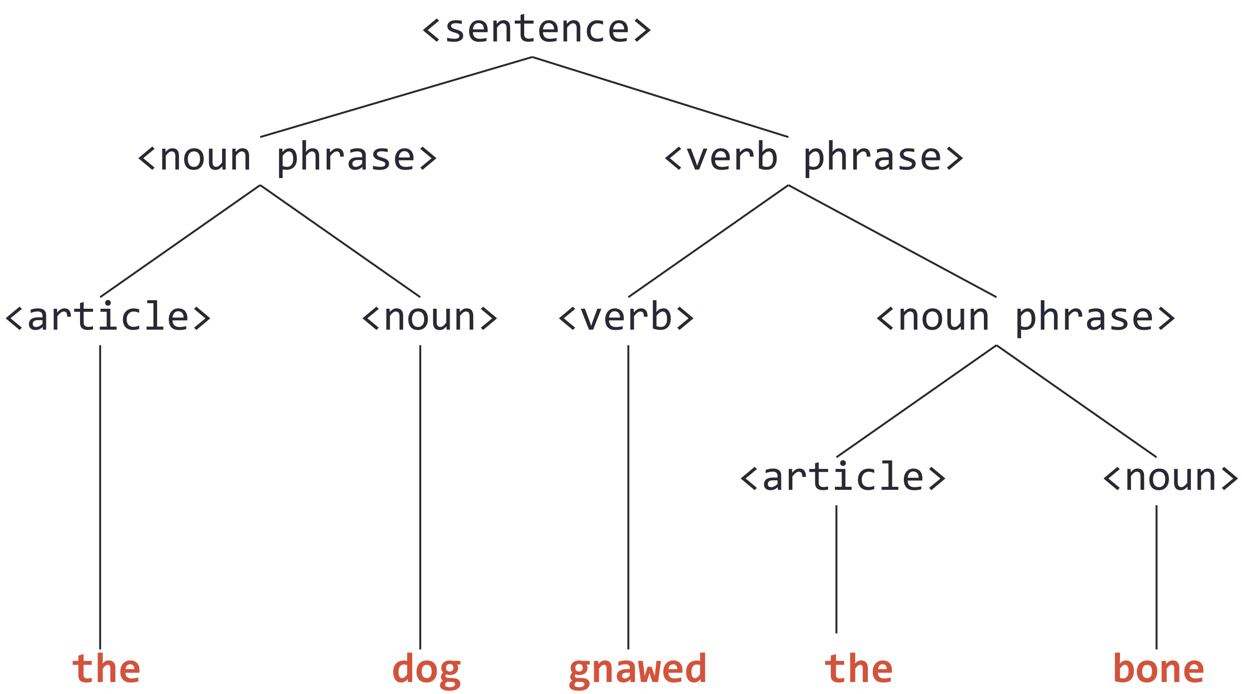
Noam Chomsky



Donald Knuth

Example

```
<sentence>    ::= <noun phrase><verb phrase>
<noun phrase> ::= article noun
<verb phrase> ::= verb <noun phrase>
```



Syntax and semantics

- **Syntax**: defines ***how*** valid sentences are **formed**
- **Semantics**: defines the ***meaning*** of valid sentences
- Some grammatically correct sentences can have no meaning
  - *"The bone walked the dog"*
- It is impossible to automatically validate the full meaning of all syntactically valid English sentences
  - Spoken languages may have ambiguous meaning
  - Programming languages must be non-ambiguous
- In programming languages, semantics is about giving a meaning by translating programs into executables

Grammars

- A grammar is a quadruple (T, N, S, R)
  - T: a finite set of terminal symbols
  - N: a finite set of non-terminal symbols
  - S: a unique starting symbol $(S \in N)$
  - R: a finite set of productions
    - $\alpha \rightarrow \beta \mid (\alpha, \beta \in (T \cup N)^*)$

- ***Context free*** grammars have productions of the form:
  - $A \rightarrow \beta \mid (A \in N) \wedge (\beta \in (T \cup N)^*)$

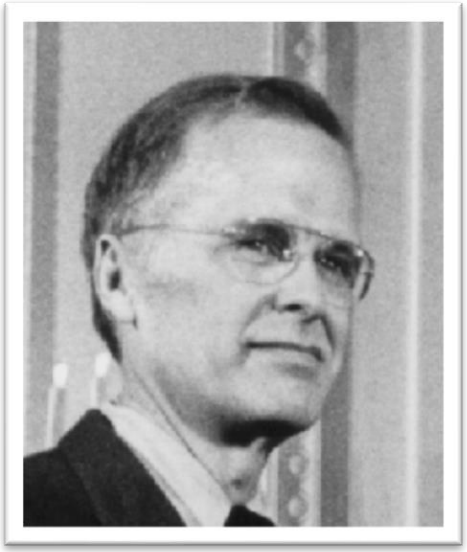
- $\alpha \mid \alpha \in (T \cup N)^*$ is called a ***sentential form***:
  - `the dog <verb> the bone`
  - `gnawed bone <noun> the`
- $\alpha \mid \alpha \in (T)^*$ is called a ***sentence:***
  - `the dog gnawed the bone`
  - `gnawed bone the the`

Backus-Naur Form

- J.W. Backus: main designer of the first FORTRAN compiler
- P. Naur: main designer of the Algol-60 programming language

  - non-terminals are placed in angle brackets
  - the symbol ::= is used instead of an arrow
  - a vertical bar can be used to signify **alternatives**
  - curly braces are used to signify an indefinite number of **repetitions**
  - square brackets are used to signify **optionality**

- Widely used to represent programming languages' syntax
- Meta-language



John Backus

Peter Naur

BNF: Example

- Pascal type declarations
  - Grammar in BNF:

```
<typedecl>      ::= type <typedeflist>
<typedeflist>   ::= <typedef> [ <typedeflist> ]
<typedef>       ::= <typeid> = <typespec> ;
<typespec>      ::= <typeid>
                  | <arraydef>
                  | <ptrdef>
                  | <rangedef>
                  | <enumdef>
                  | <recdef>
<typeid>        ::= id
<arraydef>      ::= [ packed ] array <lbrack> <rangedef> <rbrack> of <typeid>
<lbrack>        ::= [
<rbrack>        ::= ]
<ptrdef>        ::= ^<typeid>
<rangedef>      ::= <number> .. <number>
<number>        ::= <digit> [ <number> ]
<enumdef>       ::= <lparen> <idlist> <rparen>
<lparen>        ::= (
<rparen>        ::= )
<idlist>        ::= <ident> { , <ident> }
<recdef>        ::= record <vardecllist> end ;
<vardecllist>   ::= <vardecl> [ <vardecllist> ]
<vardecl>       ::= <idlist> : <typespec> ;
```

  - Example:

```
type string20 = packed array[1..20] of char;
type intptr   = ^integer;
     floatptr = ^real;
type herb     = (tarragon, rosemary, thyme, alpert);
     tinyint  = 1..7;
     student  = record
                  name, address : string20;
                  studentid     : array[1..20] of integer;
                  grade         : char;
                end;;
```

Example

- Grammar for simple arithmetic expressions:

$$
\begin{aligned}
G &= (T, N, S, R),\\
T &= \{\mathbf{id}, +, -, *, /, (,)\},\\
N &= \{E\},\\
S &= E,\\
R &= \{E \rightarrow E + E,\\
&\quad\ E \rightarrow E - E,\\
&\quad\ E \rightarrow E * E,\\
&\quad\ E \rightarrow E\ /\ E,\\
&\quad\ E \rightarrow (\ E\ ),\\
&\quad\ E \rightarrow \mathbf{id}\}
\end{aligned}
$$

Example

- Parse the sequence: **`(a+b)/(a–b)`**
- The lexical analyzer tokenizes the sequence as: **`(id+id)/(id–id)`**
- Construct a **parse tree** for the expression:
  - start symbol = root node
  - non-terminal = internal node
  - terminal = leaf
  - production, sentential form = subtree
  - sentence = tree

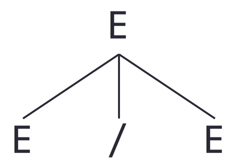
Top-down parsing

- Starts at the root (starting symbol)
- Builds the tree downwards from:
  - the sequence of tokens in input (from left to right)
  - the rules in the grammar

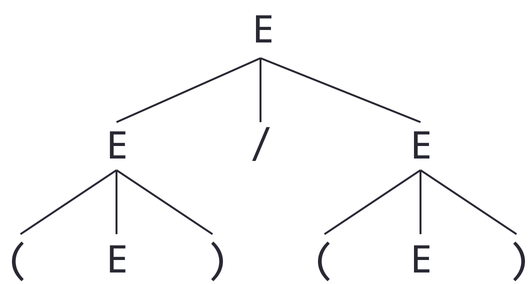
Example

(id+id)/(id–id)

1- Using: E → E / E

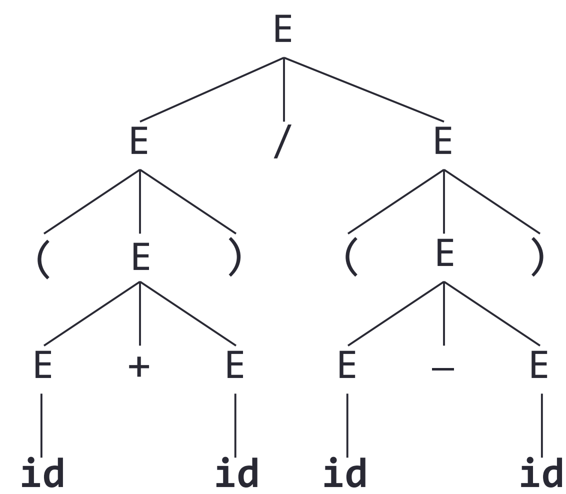


2- Using: E → (E)



E → E + E
E → E – E
E → E * E
E → E / E
E → (E)
E → **id**

3- Using: E → E + E
E → E – E
E → **id**

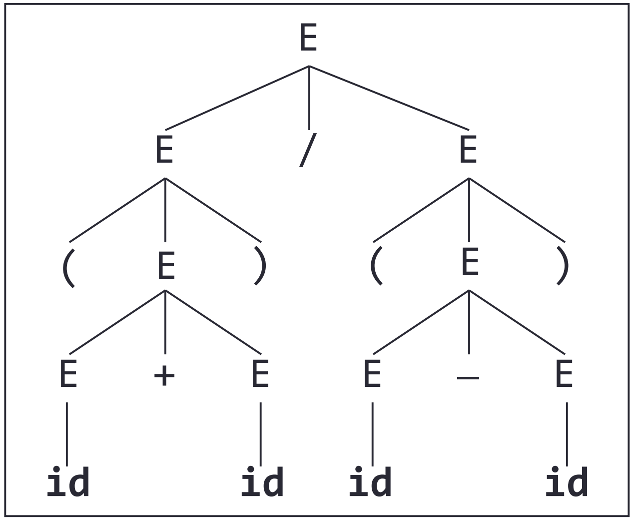


Derivations

- The application of grammar rules towards the recognition of a grammatically valid sequence of terminals can be represented with a ***derivation***
- Noted as a series of transformations:
  - $\{\alpha \Rightarrow \beta\ [\rho] \mid (\alpha,\beta \in (T \cup N)^*) \wedge (\rho \in R)\}$
  - where production ρ is used to transform α into β.

Derivation example

```
E ⇒ E / E                                [E → E / E]
  ⇒ E / ( E )                            [E → ( E )]
  ⇒ E / ( E – E )                        [E → E – E]
  ⇒ E / (E – id )                        [E → id]
  ⇒ E / ( id – id )                      [E → id]
  ⇒ ( E ) / ( id – id )                  [E → ( E )]
  ⇒ ( E + E ) / ( id – id )              [E → E + E]
  ⇒ ( E + id ) / ( id – id )             [E → id]
  ⇒ ( id + id ) / ( id – id )            [E → id]
```



- In this case, we say that E $\overset{*}{\underset{G}{\Rightarrow}}$ **(id+id)/(id–id)**
- The ***language*** generated by the grammar can be defined as:
  - $L(G) = \{\omega \mid S \overset{*}{\underset{G}{\Rightarrow}} \omega \wedge \omega \in (T)^*\}$

Leftmost and rightmost derivation

Leftmost Derivation

```
E ⇒ E / E                                  [E → E / E]
  ⇒ ( E ) / E                              [E → ( E )]
  ⇒ ( E + E ) / E                          [E → E + E]
  ⇒ ( id + E ) / E                         [E → id]
  ⇒ ( id + id ) / E                        [E → id]
  ⇒ ( id + id ) / ( E )                    [E → ( E )]
  ⇒ ( id + id ) / ( E – E )                [E → E – E]
  ⇒ ( id + id ) / ( id – E )               [E → id]
  ⇒ ( id + id ) / ( id – id )              [E → id]
```

Rightmost Derivation

```
E ⇒ E / E                                  [E → E / E]
  ⇒ E / ( E )                              [E → ( E )]
  ⇒ E / ( E – E )                          [E → E – E]
  ⇒ E / (E – id )                          [E → id]
  ⇒ E / ( id – id )                        [E → id]
  ⇒ ( E ) / ( id – id )                    [E → ( E )]
  ⇒ ( E + E ) / ( id – id )                [E → E + E]
  ⇒ ( E + id ) / ( id – id )               [E → id]
  ⇒ ( id + id ) / ( id – id )              [E → id]
```

Top-down and bottom-up parsing

- A **top-down** parser builds a parse tree starting at the root down to the leafs
  - It builds *leftmost* derivations, i.e. a forward derivation proving that a sentence can be generated from the starting symbol by using a sequence of *forward* applications of productions:

```
E ⇒ E / E                                  [E → E / E]
  ⇒ ( E ) / E                              [E → ( E )]
  ⇒ ( E + E ) / E                          [E → E + E]
  ⇒ ( id + E ) / E                         [E → id]
  ⇒ ( id + id ) / E                        [E → id]
  ⇒ ( id + id ) / ( E )                    [E → ( E )]
  ⇒ ( id + id ) / ( E – E )                [E → E – E]
  ⇒ ( id + id ) / ( id – E )               [E → id]
  ⇒ ( id + id ) / ( id – id )              [E → id]
```

- A **bottom-up** parser builds a parse tree starting from the leafs up to the root
  - It builds *rightmost* derivations, i.e. a reverse derivation proving that one can come to the starting symbol from a sentence by applying a sequence of *reverse* applications of productions:

```
  ⇐ ( id + id ) / ( id – id )              [E → id]
  ⇐ ( E + id ) / ( id – id )               [E → id]
  ⇐ ( E + E ) / ( id – id )                [E → ( E + E )]
  ⇐ ( E ) / ( id – id )                    [E → ( E )]
  ⇐ E / ( id – id )                        [E → id]
  ⇐ E / ( E – id )                         [E → id]
  ⇐ E / ( E – E )                          [E → E - E]
  ⇐ E / ( E )                              [E → ( E )]
E ⇐ E / E                                  [E → E / E]
```

Grammar transformations

Tranforming extended BNF grammar constructs

- Extended BNF includes constructs for **optionality** and **repetition.**
- They are very convenient for clarity/conciseness of presentation of the grammar.
- However, they have to be removed, as they are not compatible with standard generative parsing techniques.

Transforming optionality and repetition

- For **optionality** BNF constructs:

1- Isolate productions of the form:

$$A \rightarrow \alpha [X_1 \ldots X_n] \beta \qquad \text{(optionality)}$$

2- Introduce a new non-terminal **N**

3- Introduce a new rule

$$A \rightarrow \alpha \; N \; \beta$$

4- Introduce two rules to generate the optionality of **N**

$$N \rightarrow X_1 \ldots X_n$$
$$N \rightarrow \varepsilon$$

- For **repetition** BNF constructs:

1- Isolate productions of the form:

$$A \rightarrow \alpha \{X_1 \ldots X_n\} \beta \qquad \text{(repetition)}$$

2- Introduce a new non-terminal **N**

3- Introduce a new rule

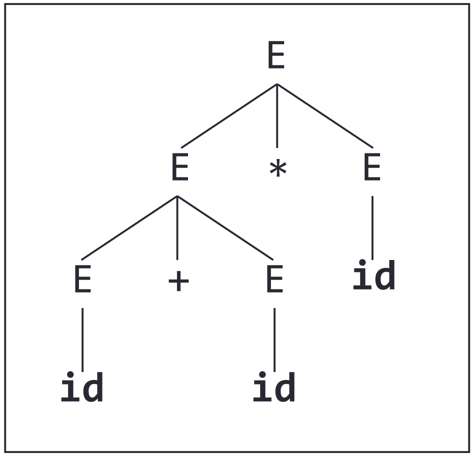
$$A \rightarrow \alpha \; N \; \beta$$

4- Introduce two rules to generate the repetition of **N**

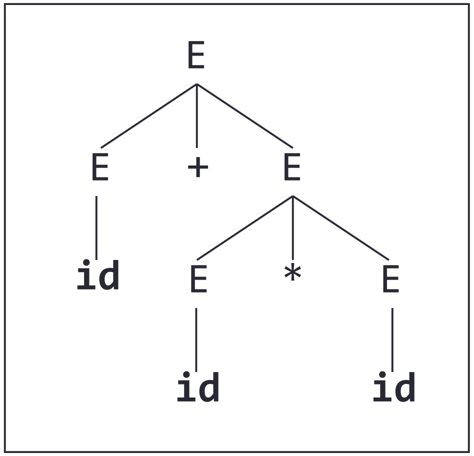
$$N \rightarrow X_1 \ldots X_n \; N \qquad \text{(right recursion)}$$
$$N \rightarrow \varepsilon$$

Ambiguous grammars

- Which of these trees is the right one for the expression "**id + id * id**" ?



E → E + E
E → E – E
E → E * E
E → E / E
E → (E)
E → **id**



- According to the grammar, both are right.
- The language defined by this grammar is ***ambiguous***.
- That is not acceptable in a compiler.
- Non-determinism needs to be avoided.

Removing ambiguities

- Solutions:
  - Incorporate operation precedence in the parser (complicates the compiler, rarely done)
  - Implement backtracking (complicates the compiler, inefficient)
  - Transform the grammar to remove ambiguities

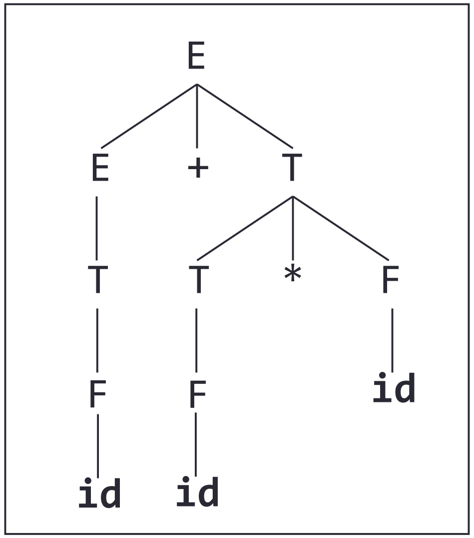
  - Example: introduce operator precedence in the grammar

```
E → E + E
E → E – E
E → E * E
E → E / E
E → ( E )
E → id
```

→

```
E → E + T
E → E – T
E → T
T → T * F
T → T / F
T → F
F → ( E )
F → id
```

→



  - Example: factorization

```
A → aB
A → aC
A → b
```

→

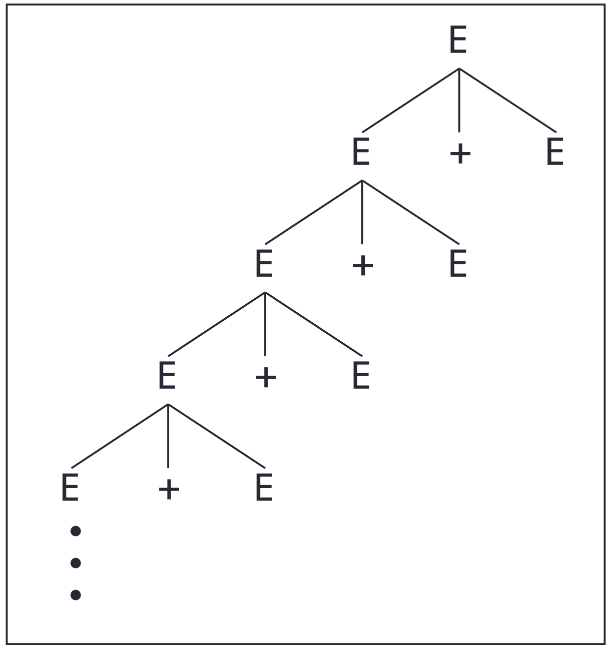
```
A → aD
A → b
D → B
D → C
```

Left recursion

- The aim is to design a parser that has no arbitrary choices to make between rules (*predictive parsing*)
- In predictive parsing, the assumption is that the first rule that can apply is applied, as there are never two different applicable rules.
- In this case, productions of the form **A→Aα** will be applied forever

id + id + id

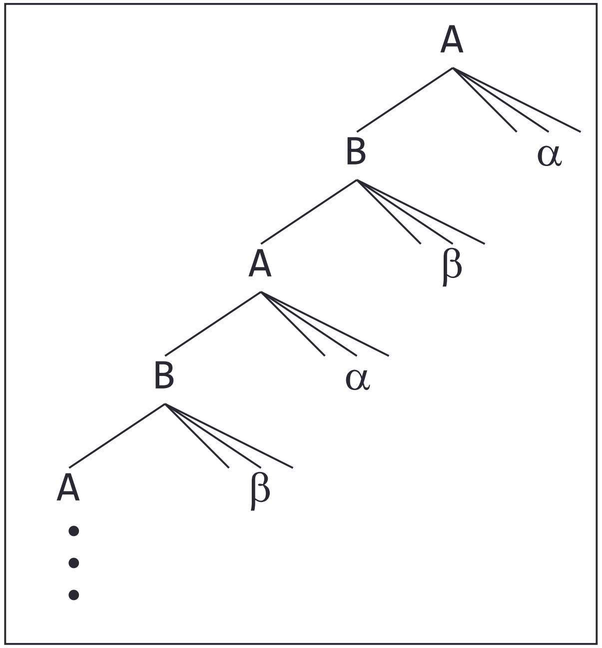
```
E → E + E
E → E – E
E → E * E
E → E / E
E → ( E )
E → id
```



Non-immediate left recursion

- Left recursions may seem to be easy to locate.
- However, they may be **transitive**, or **non-immediate**.
- Non-immediate left recursions are sets of productions of the form:

$$A \rightarrow B\alpha \mid \ldots$$
$$B \rightarrow A\beta \mid \ldots$$



Transforming left recursion

- This problem afflicts all top-down parsers
- **Solution**: apply a transformation to the grammar to remove the left recursions

1- Isolate each set of productions of the form:

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots$ (left-recursive)

$A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$ (non-left-recursive)

2- Introduce a new non-terminal **A′**

3- Change all the non-recursive productions on **A** to:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$

4- Remove the left-recursive production on **A** and substitute:

$A' \rightarrow \varepsilon \mid \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots$ (right-recursive)

Example

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid \textbf{id}$$

(i) $E \rightarrow E + T \mid E - T \mid T$

1- $E \rightarrow E + T \mid E - T$ $\quad (A \rightarrow A\alpha_1 \mid A\alpha_2)$

$E \rightarrow T$ $\quad (A \rightarrow \beta_1)$

2- E' $\quad (A')$

3- $E \rightarrow TE'$ $\quad (A \rightarrow \beta_1 A')$

4- $E' \rightarrow \varepsilon \mid +TE' \mid -TE'$ $\quad (A' \rightarrow \varepsilon \mid \alpha_1 A' \mid \alpha_2 A')$

$$E \rightarrow TE'$$
$$E' \rightarrow \varepsilon \mid +TE' \mid -TE'$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid \textbf{id}$$

1- Isolate each set of productions of the form:

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots$ (left-recursive)

$A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$ (non-left-recursive)

2- Introduce a new non-terminal **A′**

3- Change all the non-recursive productions on **A** to:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$

4- Remove the left-recursive production on **A** and substitute:

$A' \rightarrow \varepsilon \mid \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots$ (right-recursive)

Example

$E \rightarrow TE'$
$E' \rightarrow \varepsilon \mid +TE' \mid -TE'$
$T \rightarrow T * F \mid T / F \mid F$
$F \rightarrow (E) \mid \textbf{id}$

(ii) $T \rightarrow T * F \mid T / F \mid F$

1-	$T \rightarrow T * F \mid T / F$	$(A \rightarrow A\alpha_1 \mid A\alpha_2)$
	$T \rightarrow F$	$(A \rightarrow \beta_1)$
2-	T'	(A')
3-	$T \rightarrow FT'$	$(A \rightarrow \beta_1 A')$
4-	$T' \rightarrow \varepsilon \mid *FT' \mid /FT'$	$(A' \rightarrow \varepsilon \mid \alpha_1 A' \mid \alpha_2 A')$

$E \rightarrow TE'$
$E' \rightarrow \varepsilon \mid +TE' \mid -TE'$
$T \rightarrow FT'$
$T' \rightarrow \varepsilon \mid *FT' \mid /FT'$
$F \rightarrow (E) \mid \textbf{id}$

1- Isolate each set of productions of the form:
$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots$ (left-recursive)
$A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$ (non-left-recursive)
2- Introduce a new non-terminal **A′**
3- Change all the non-recursive productions on **A** to:
$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$
4- Remove the left-recursive production on **A** and substitute:
$A' \rightarrow \varepsilon \mid \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots$ (right-recursive)

Non-recursive ambiguity

- As the parse is essentially predictive, it cannot be faced with non-deterministic choice as to what rule to apply
- There might be sets of rules of the form: $\mathbf{A} \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots$
- This would imply that the parser needs to make a choice between different right hand sides that begin with the same symbol, which is not acceptable
- They can be eliminated using a factorization technique

1- Isolate a set of productions of the form:

$$\mathbf{A} \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \qquad \text{(ambiguity)}$$

2- Introduce a new non-terminal **A′**

3- Replace all the ambiguous set of productions on **A** by:

$$\mathbf{A} \rightarrow \alpha\mathbf{A'} \qquad \text{(factorization)}$$

4- Add a set of factorized productions on **A′** :

$$\mathbf{A'} \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$$

Predictive parsing

Backtracking

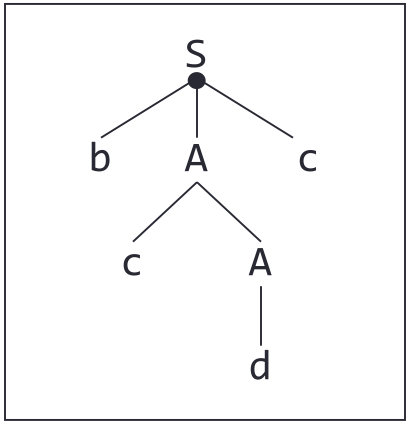
- It is possible to write a parser that implements an ambiguous grammar.
- In this case, when there is an arbitrary alternative, the parser explores the alternatives one after the other.
- If an alternative does not result in a valid parse tree, the parser backtracks to the last arbitrary alternative and selects another right-hand-side.
- The parse fails only when there are no more alternatives left .
- This is often called a **brute-force method**.

Example

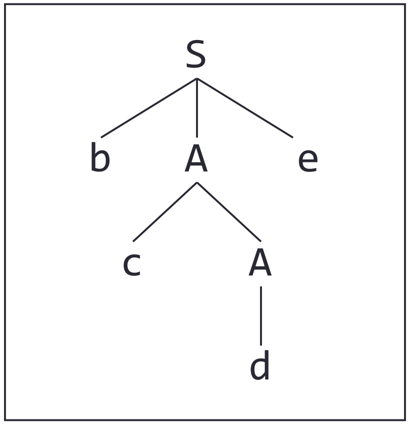
S → ee | bAc | bAe
A → d | cA

Seeking for : bcde

S	⇒ bAc	[S → bAc]•
	⇒ bcAc	[A → cA]
	⇒ bcdc	[A → d]
	⇒ **error**	



S	⇒ bAe	[S → bAe]
	⇒ bcAe	[A → cA]
	⇒ bcde	[A → d]
	⇒ **OK**	



Backtracking

- Backtracking is tricky and inefficient to implement.
- Generally, code is generated as rules are applied; backtracking involves retraction of the generated code!
- Parsing with backtracking is seldom used.
- The most simple solution is to eliminate the ambiguities from the grammar.
- Some more elaborated solutions have been recently found that optimize backtracking that use a **caching** technique to **reduce the number of generated sub-trees** [2,3,4,5].

Predictive parsing

- **Restriction**: the parser must always be able to determine which of the right-hand sides to follow, only with its knowledge of the next token in input.
- Top-down parsing without backtracking.
- Deterministic parsing.
- The assumption is that no backtracking is possible/necessary.

Predictive parsing

- <u>**Recursive descent predictive parser**</u>
  - A **function** is defined for **each non-terminal symbol**.
  - Its predictive nature allows it to choose the right right-hand-side.
  - It recognizes terminal symbols and calls other functions to recognize non-terminal symbols in the chosen right hand side.
  - The parse tree is actually constructed by the nesting of function calls.
  - Very easy to implement.
  - Hard-coded: allows to handle unusual situations.
  - Hard to maintain.

Predictive parsing

- Table-driven predictive parser
  - A ***parsing table*** tells the parser which right-hand-side to choose.
  - The ***driver algorithm*** is standard to all parsers.
  - Only the table changes when the language changes, the algorithm is universal.
  - Easy to maintain.
  - The parsing table is hard and error-prone to build for most languages.
  - Tools can be used to generate the parsing table.
  - Will be covered in next lecture.

First and Follow sets

First and Follow sets

- When parsing using a certain non-terminal symbol, predictive parsers need to know what right-hand-side to choose, knowing only what is the next token in input.
- If all the right hand sides begin with terminal symbols, the choice is straightforward.
- If some right hand sides begin with non-terminals, the parser must know what token can begin any sequence generated by this non-terminal (i.e. the FIRST set of these non-terminals).
- If a FIRST set contains ε, it must know what may follow this non-terminal (i.e. the FOLLOW set of this non-terminal) in order to chose an ε production.

Example

```
E   → TE'
E'  → +TE' | ε
T   → FT'
T'  → *FT' | ε
F   → 0 | 1 | (E)
```

```
FIRST(E)   = {0,1,(}
FIRST(E')  = {+, ε}
FIRST(T)   = {0,1,(}
FIRST(T')  = {*, ε}
FIRST(F)   = {0,1,(}
```

```
FOLLOW(E)  = {$,)}
FOLLOW(E') = {$,)}
FOLLOW(T)  = {+,$,)}
FOLLOW(T') = {+,$,)}
FOLLOW(F)  = {*,+,$,)}
```

Example: Recursive descent predictive parser

```
E  → TE'
E' → +TE' | ε
T  → FT'
T' → *FT' | ε
F  → 0 | 1 | (E)
```

```
FIRST(E)  = {0,1,(}
FIRST(E') = {+, ε}
FIRST(T)  = {0,1,(}
FIRST(T') = {*, ε}
FIRST(F)  = {0,1,(}
```

```
FOLLOW(E)  = {$,)}
FOLLOW(E') = {$,)}
FOLLOW(T)  = {+,$,)}
FOLLOW(T') = {+,$,)}
FOLLOW(F)  = {*,+,$,)}
```

```
error = false
Parse(){
  lookahead = NextToken()
  if (E();match('$')) return true
  else return false}
E(){
  if (lookahead is in [0,1,(])        //FIRST(TE')
    if (T();E'();)
      write(E->TE')
    else error = true
  else error = true
  return !error}
E'(){
  if (lookahead is in [+])            //FIRST[+TE']
    if (match('+');T();E'())
      write(E'->TE')
    else error = true
  else if (lookahead is in [$,)]      //FOLLOW[E'] (epsilon)
    write(E'->epsilon)
  else error = true
  return !error}
T(){
  if (lookahead is in [0,1,(])        //FIRST[FT']
    if (F();T'();)
      write(T->FT')
    else error = true
  else error = true
  return !error}
```

Example: Recursive descent predictive parser

```
E  → TE'
E' → +TE' | ε
T  → FT'
T' → *FT' | ε
F  → 0 | 1 | (E)
```

```
FIRST(E)   = {0,1,(}
FIRST(E')  = {+, ε}
FIRST(T)   = {0,1,(}
FIRST(T')  = {*, ε}
FIRST(F)   = {0,1,(}
```

```
FOLLOW(E)  = {$,)}
FOLLOW(E') = {$,)}
FOLLOW(T)  = {+,$,)}
FOLLOW(T') = {+,$,)}
FOLLOW(F)  = {*,+,$,)}
```

```
T'(){
  if (lookahead is in [*])              //FIRST[*FT']
    if (match('*');F();T'())
      write(T'->*FT')
    else error = true
  else if (lookahead is in [+,),$]      //FOLLOW[T'] (epsilon)
    write(T'->epsilon)
  else error = true
  return !error}
F(){
  if (lookahead is in [0])              //FIRST[0]
    match('0');write(F->0)
  else if (lookahead is in [1])         //FIRST[1]
    match('1');write(F->1)
  else if (lookahead is in [(])         //FIRST[(E)]
    if (match('(');E();match(')'))
      write(F->(E));
    else error = true
  else error = true
  return !error}
}
```

References

1. C.N. Fischer, R.K. Cytron, R.J. LeBlanc Jr., "Crafting a Compiler", Adison-Wesley, 2009. Chapter 4.
2. Frost, R., Hafiz, R. and Callaghan, P. (2007) " Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars ." *10th International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE* , Pages: 109-120, June 2007, Prague.
3. Frost, R., Hafiz, R. and Callaghan, P. (2008) "Parser Combinators for Ambiguous Left-Recursive Grammars." *10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN* , Volume 4902/2008, Pages: 167-181, January 2008, San Francisco.
4. Frost, R. and Hafiz, R. (2006) "A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time." *ACM SIGPLAN Notices*, Volume 41 Issue 5, Pages: 46 - 54.
5. Norvig, P. (1991) "Techniques for automatic memoisation with applications to context-free parsing." *Journal - Computational Linguistics.* Volume 17, Issue 1, Pages: 91 - 98.
6. DeRemer, F.L. (1969) "Practical Translators for LR(k) Languages." PhD Thesis. MIT. Cambridge Mass.

References

7. DeRemer, F.L. (1971) “Simple LR(k) grammars.” Communications of the ACM. 14. 94-102.
8. Earley, J. (1986) “An Efficient Context-Free Parsing Algorithm.” PhD Thesis. Carnegie-Mellon University. Pittsburgh Pa.
9. Knuth, D.E. (1965) “On the Translation of Languages from Left to Right.” Information and Control 8. 607-639. doi:10.1016/S0019-9958(65)90426-2
10. Dick Grune; Ceriel J.H. Jacobs (2007). “Parsing Techniques: A Practical Guide.” Monographs in Computer Science. Springer. ISBN 978-0-387-68954-8.
11. Knuth, D.E. (1971) “Top-down Syntax Analysis.” Acta Informatica 1. pp79-110. doi: 10.1007/BF00289517