CPSC 323: Compilers and Languages

Chapter 3 – Syntactic Analysis

Mr. Param Venkat Vivek Kesireddy
Teaching Associate
Mail: pkesireddy@fullerton.edu

Inputs from Prof Doina Bein & Prof James Choi

Chapter 3. Syntax Analysis I (Top-down parsing)

1. Introduction
2. Grammar
3. Chomsky Hierarchy
4. Left-Recursion and Back-Tracking
5. Top-Down Parsers
   a. Recursive Descent Parser (RDP)
   b. Predictive Recursive Descent Parser (PRDP)
   c. Table Driven Predictive Parser

3.1 Introduction

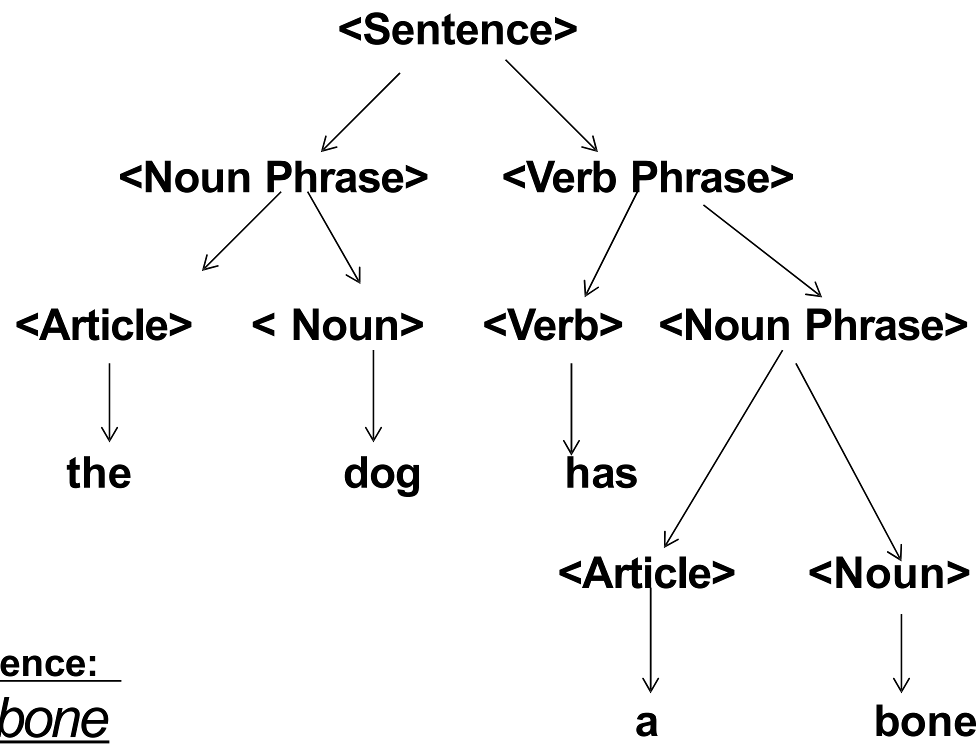
Syntax Analysis is a phase where the structure of source code is recognized and constructed using a finite set of rules called productions.

Ex. of Rules for a subset of English

<Sentence> -> <Noun Phrase> <Verb Phrase>
<Noun Phrase> -> <Article> <Noun>
<Verb Phrase> -> <Verb> <Noun Phrase>
<Noun> -> dog, cat, bone, school
<Article> -> the, a, an
<Verb> -> goes, has

Terms:
Production, Non-terminal Symbols, Terminal symbols

Sentence: The dog has a bone



However, the 2nd sentence:

The dog goes a bone

has no meaning!

BNF NOTATION: Backus-Naur(Normal) Form

A Specific notation to describe the productions.

Ex.

<Expression> ::= <Expression> + <Term> |
<Expression> - <term> | <Term>

<Term> ::= <Term> * <Factor> | <Term> / <Factor> |
<Factor>

<Factor> ::= <Identifier> | <Number> | (<Expression>)

Extended BNF: Allows additional notations; such as
{ } 0 or more times,
[ ] optional

<Number> ::= <Digit>
{ <Digit> }
<Identifier> ::=
<Letter> { <Letter> |
<Digit> }
<Factor> ::=
[-] <Primary>

CALIFORNIA STATE UNIVERSITY
FULLERTON

3.2 Grammar

a) <u>Def:</u> <u>Grammar</u> G = (T, N, S, R) where

T is a finite set of terminal symbols

N is a finite set of non-terminal symbols

$S \in N$ is a unique Starting symbol

R is a finite set of productions of the form

$\alpha \rightarrow \beta$ where

α, β are strings of terminal and non-terminal symbols

Def: <u>The language of grammar G</u> is the set of all sentences that can be generated by G, and it is written L{G}.

Ex. of a Grammar

G = (

T = {id, +, -, *, /,
(,) },

N = {<Expr>, <Term>, <Factor> } S =
<Expr>,

R = { 1. <Expr> -> <Expr> +
<Term>

2. <Expr> -> <Expr> - <Term>
3. <Expr> -> <Term>
4. <Term> -> <Term> * <Factor>
5. <Term> -> <Term> / <Factor>
6. <Term> -> <Factor>
7. <Factor> -> id
8. <Factor> -> (<Expr>)

}

)

b) Derivation

Def: Derivation is replacing one non-terminal symbol at a time in order to recognize a sentence.

Ex. Source code: ***a / (c – d)***

<Expr> => <term>

=> <Term> / <Factor>

⇒ <Factor> / <Factor>

⇒ id(a) / <Factor>

⇒ a / (<Expr>)

⇒ a / (<Expr> - <Term>)

⇒ a / (<Term> - <Term>)

⇒ a / (<Factor> - <Term>)

⇒ a / (c - <Term>)

⇒ a / (c - <Factor>)

⇒ a / (c – d)

In general, we say <Expr> => a / (c-d)

- Sentential Form: is a string of symbols (N or T) appearing in various steps in derivation.

- Left Most Derivation(LMD) : Replace the Left most Non-terminal in each step.

- Right Most Derivation(RMD): Replace the Right most non-terminal in each step.

c)Ambiguity

Def: A grammar is ambiguous if there are two different parse trees for some words in L(G).

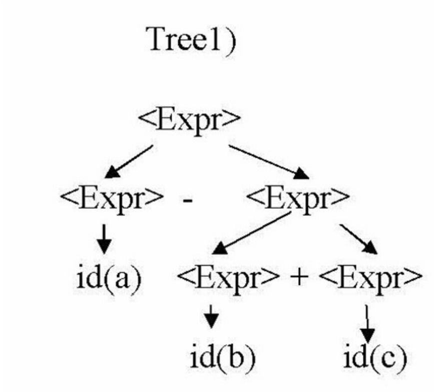
Ex: Assume Productions

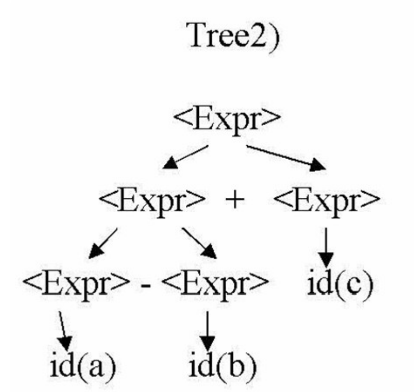
<Expr> -> <Expr> + < Expr>

<Expr> -> <Expr> - <Expr>

<Expr> -> id

Consider a string a- b+c





2 different parse trees => the grammar is ambiguous

CALIFORNIA STATE UNIVERSITY FULLERTON

Def: A language for which NO UNAMBIGUOUS grammar exists is called Inherently ambiguous language.

G1 G2

G3

….

Gn

$L(G) = \{ s1, s2, s3, \ldots\ldots\ldots s\infty \}$

All G1, G2,….,Gn are ambiguous

CALIFORNIA STATE UNIVERSITY FULLERTON

3.3 Chomsky Hierarchy of Grammars

Chomsky Hierarchy shows different types of grammars based on the forms of productions. => 4 types (0,1,2 and 3).

New Convention:

- Capital Letter: A, B …. Non-terminal
- Lowercase letters: a,b,c … terminals
- Greek Letters: α, β, δ, γ … strings of N and T

a) Type 0: Unrestricted Grammar

Def: Production form: α -> β

where both (α, β) are <u>any</u> string of N and T

Ex.

R1) S -> ACaB

R2) Ca -> aa C

R3) CB -> DB

R4) CB -> E

R5) aD -> D a

R6) AD -> AC

R7) aE -> Ea

R8) AE -> ε

A Simple Strings:
aa, aaaa,
aaaaaaaa etc

R1

R2

R4

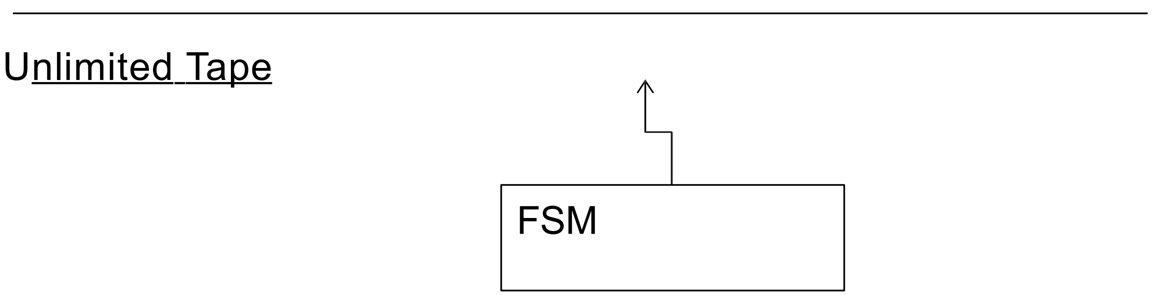
R7

R7

CALIFORNIA STATE UNIVERSITY FULLERTON

Machine for Unrestricted languages:

Turing Machine (By Allen Turing 1912-1954)



When you have a Type 0 grammar, you can use a Turing machine to accept and generate strings in the language defined by that grammar.

b) Type 1: Context Sensitive Grammar

Def: production Form: $\gamma A\delta \to \gamma\alpha\delta$,

Similar to type 0 but

1) α cannot be ε i.e., -> $|\gamma\alpha\delta|$ >= $|\gamma A\delta|$, RHS >= LHS

2) A -> α in the context of γ, δ.

Ex.

R1) S -> aSBC

R2) S -> abC

R3) CB -> BC /* Strictly speaking is not context sensitive */

R4) bB -> bb

R6) bC -> bc

R6) cC -> cc

Sample string:

abc, aabbcc

etc

R1

R2

R3

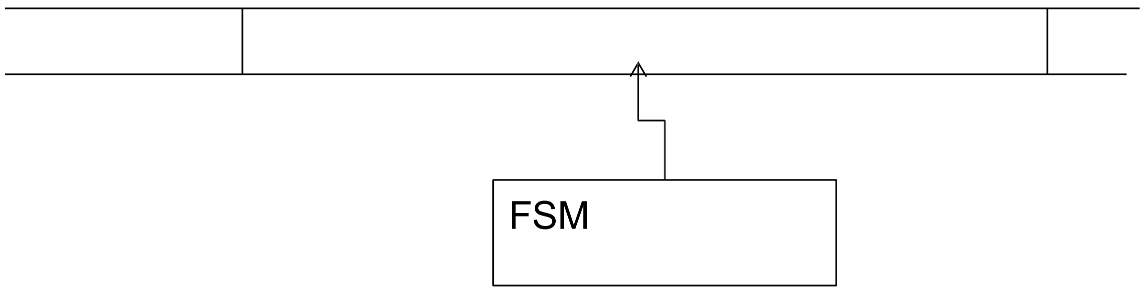
R4

CALIFORNIA STATE UNIVERSITY FULLERTON

Machine for Context Sensitive languages:

LBA (Lineary Bounded Automaton): Similar to Turing but with

limited Tape



For recognizing and generating languages defined by context-sensitive grammars, which are a level above context-free grammars in the Chomsky Hierarchy, Linear Bounded Automata (LBA) are the appropriate computational devices.

CALIFORNIA STATE UNIVERSITY FULLERTON

c) Type 2: Context Free Grammar

Def: Production Form: $\alpha \rightarrow \beta$

where α is a single non terminal

Ex.

R1) S -> a B

R2) S -> bA

R3) A -> a

R4) A -> aS

R5) A -> bAA

R6) B -> b

R7) B -> bS

R8) B -> a BB

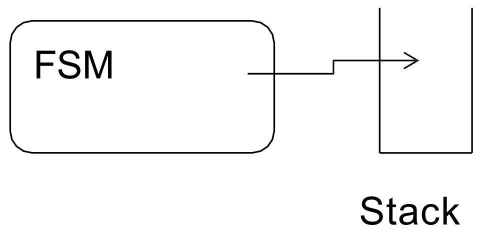
Simple String:

ab, ba, abab,
aaabbb etc

L(G) ={ w | w has same number of a's and b's}

CALIFORNIA STATE UNIVERSITY FULLERTON

c) Machine: Push Down Automaton (PDA)



- With the inclusion of a stack, a pushdown automaton (PDA) is a sort of finite automata that enhances the capabilities of a finite automaton.
- PDAs have a balance that makes them appropriate for a variety of real-world language processing and parsing applications.

d) Type 3: Regular Grammar

Def: α -> β where

α is a single nonterminal and,

β is either all terminals or at most one Non-terminal

(the last symbol on the RHS, first or last symbol)

Ex. R1) S -> bA

R2) S -> aB

R3) S -> ε

R4) A -> abaS

R5) B -> babS

String: abab, baba, abababab, babaabab etc

(abab | baba)* which is an RE

<u>Machine: FSM</u>

CALIFORNIA STATE UNIVERSITY FULLERTON

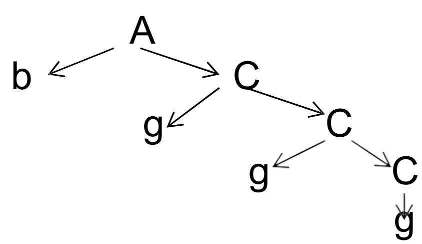
3.4 Left-Recursion and Back-Tracking

Q: How do we build a Parser for CFL?

Initially let's briefly look at again how a parser works.

Ex. A -> Ba | bC
B -> d | eBF
C -> gC | g

Let's Consider a string bggg



- **Starts from the Top of the Tree and continues to parse until all token are matched. (Top-Down Parser)**

- **Now before we build a such parser from the grammar, we have to consider 2 issues**

CALIFORNIA STATE UNIVERSITY FULLERTON

a) Left Recursion

We have to eliminate all left-recursive productions in the grammar.

A left-recursive production is when the LHS of a production occurs as the first symbol on the RHS of the same production

ex. E -> E + T

Why?

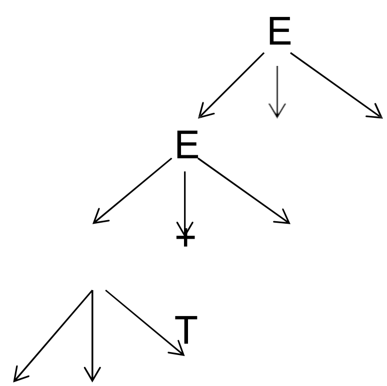
Let's consider the following rules:

R1) E -> E + T

R2) E -> T

R3) T -> id

Given string **a + b + c ,** the parser tries to match.



=> the parser does not know when to stop expanding E -> E +T

Since it does not know how many + the string has

Q: How do we eliminate left recursions?

A: By changing the productions as follows:

Assume we have the following productions.

$A \rightarrow A\alpha$

$A \rightarrow \delta$

Steps:

1. Introduce a new Non-terminal A'

2.Change $A \rightarrow \delta A'$

3. $A' \rightarrow \alpha A' \mid \varepsilon$

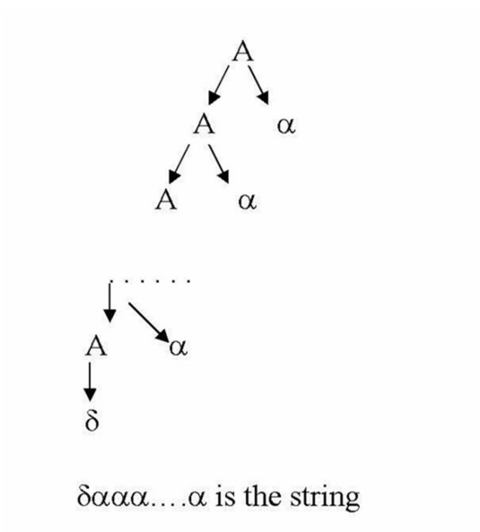
=>

$A \rightarrow \delta A$

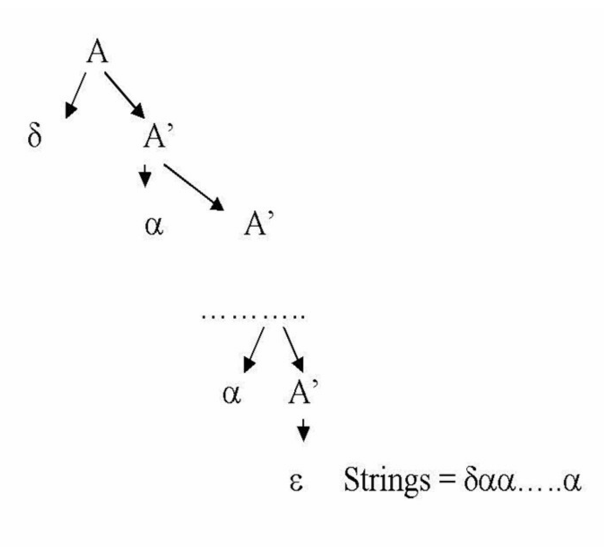
$A' \rightarrow \alpha A' \mid \varepsilon$

Let’s See what we have done

A -> Aα
A -> δ



A -> δ A'
A’ -> α A’ | ε



Note: We have two parser trees with same strings but the shapes are different

Ex. of Removing Left recursion

E -> E +

T E -> T

T -> id

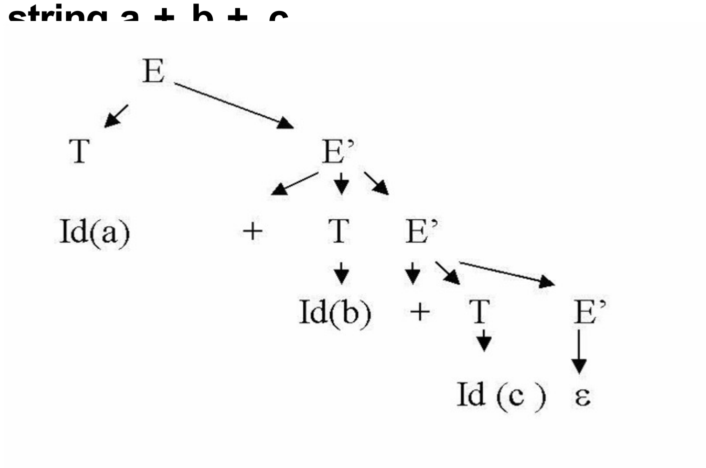
Remove left recursion =>

E -> T E’

E’ -> +TE’ | ε

T -> id

Let’s further consider the string a + b + c



<u>But there can be "indirect (non-immediate)" left-recursion.</u>

Ex.

A -> B C

B -> A

C

Thus, we need an algorithm to remove all left-recursions:

If RHS begins with a NT (A) earlier in the list then

- **Substitute A**
- **Remove any direct recursion**

Ex. A -> N | β

...

N -> Aγ

Method:

Then replace A =>

N -> Nγ | βγ (remove direct left-recursion)

1. Make a list of all NT (ex. In the

Ex.

R1) E -> E + T   R2) E -> T   R3) T -> E   R4) T -> id

1) List of NT => 1. E, 2. T

2) Direct Left-recursion in (R1) and (R2)

E -> TE’

E’ -> + TE’ | ε

3) Need to replace E in (R3)

T -> TE’

4) Here is Left-Recursion

T -> id T’

T’ -> E’T’ | ε

Therefore:

R1) E -> TE’

R2) E’ -> TE’ | ε

R3) T -> id T’

R4) T’ -> E’T’ | ε

=> E, E’, T, T’

Another Round…

b)Backtracking

Backtracking is reparsing of the same/previous tokens

Ex) Assuming the following productions:

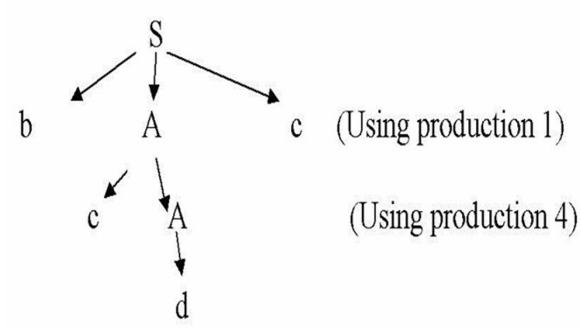
S -> b A c

S -> b A e

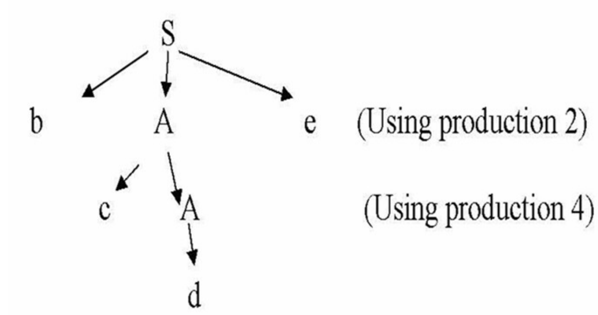
A -> d

A -> c A

and the following string



Does not work.
String wrong?
=>
Consider other Possibilities



Works!!

How to solve backtracking? =>

use of Left-Factorization =>

Factor out same symbols of RHSs of the productions for the same Non-terminal

Ex)

R1) S -> b A c

R2) S -> b A e

R3) A –-> d

R4) A -> c A

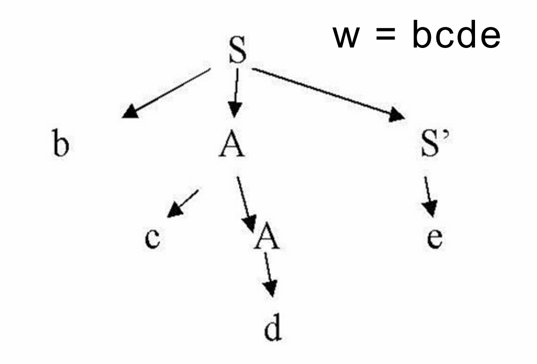
=>

R1) S -> b A S' **(Factor out bA from R1 and R2)**

R2) S' -> c | e

R3) A -> d

R4) c A



CALIFORNIA STATE UNIVERSITY FULLERTON

CONCLUSION:

1) We need to eliminate Left-Recursion
2) Remove Back-Tracking

before
constructing the Top-Down parsers

CALIFORNIA STATE UNIVERSITY FULLERTON

EX:
Given the following productions, build a PDA that accepts L(G)

R1) S -> a S a
R2) S -> b S
b R3) S -> c

Sample strings: aca, abcba,
abacaba
L(G) =
{wcw^r} (Palindrom)

PDA = (Σ = {a,b,c}, Q = {s,f}, q0 = s, F = {f}, Γ = {1,2}

N =

{ N1) (s,a,ε) -> (s,1)

N2) (s,b, ε) -> (s,2)

N3) (s,c, ε) -> (f, ε)

N4) (f,a,1) -> (f, ε)

N5) (f,b,2) -> (f, ε) }

w = abbcbba

State	Input	Stack	N-used
S	abbcbba	ε	1
S	bbcbba	ε 1	2
S	bcbba	ε21	2
S	cbba	ε221	3
f	bba	221	5
f	ba	21	5
f	a	1	4
f	ε	ε	
accepting state	empty	empty => accepted	

3.5 Top-Down Parsers

a) Recursive Descent Parser (RDP)

The basic idea is that each <u>non-terminal has an associated parsing procedure</u> that can recognize any sequence of tokens generated by that non-terminal.

For example.

1. E -> E + T
2. E -> E – T
3. E -> T
4. T -> id

1.Remove Left-Recursion

E -> TE'
E' -> + TE' | - TE' | ε
T -> id

Productions

: E -> TE'

E' -> + TE' | - TE' | ε

T -> id

Procedure E ()

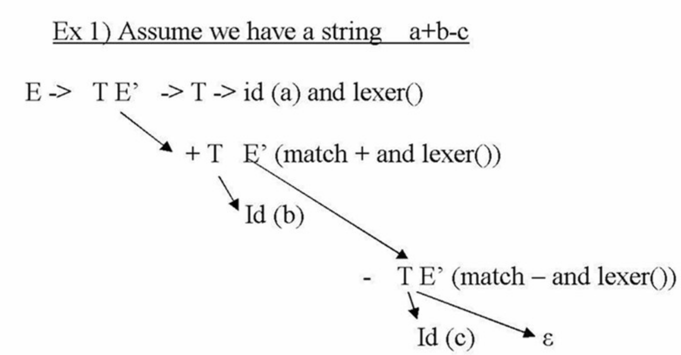
```
{
  T ();
   E'();
   If not eof marker then
       error-message
}
```

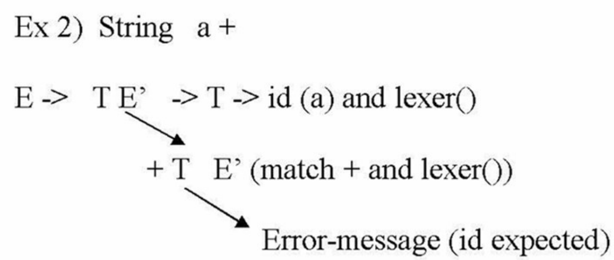
Procedure E'()

```
{
If token = + or – then
        {
        Lexer();
       T();
        E'();
        }
}
```

Procedure T();

```
{
   If token is id then
         lexer()
else error-message
    (id expected)
}
```



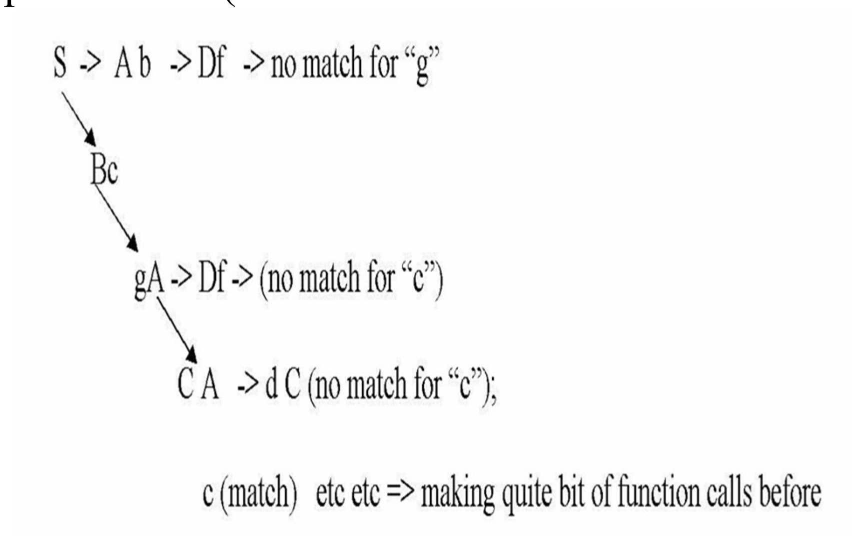


b)Predictive Recursive Descent Parser (PRDP)

A more efficient way of implementing RDP.

Assume we have the following productions (No Left-Recursion or Back-Tracking)

1. S -> Ab | B c
2. A -> Df | CA
3. B -> gA | e
4. C -> dC | c
5. D -> h | i

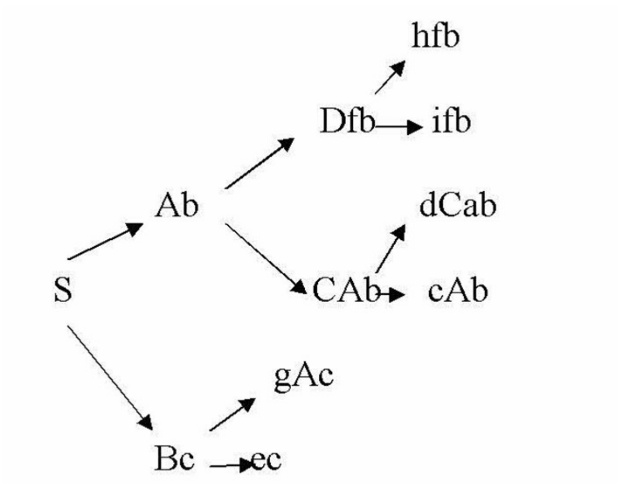


and a string "gchfc" and parse

making quite bit of function calls before matching
=> a better way? A more efficient way?

Yes: anticipate what terminal symbols are derivable from each nonterminal symbol on the RHS of productions.

Ex). Let's compute before we construct the parser.



if the parser goes (Ab) =>
it will encounter terminals {h, i, d,c}
else (BC) => {g, e}

So, before paring look ahead of the sets which way to go:
Ex. string "gchfc" => "g" is in (BC) route, so no need to go to the first route.
<u>These sets are called First sets.</u> First (Ab) = { h, i, d, c}
First (Bc) = {g, e}

CALIFORNIA STATE UNIVERSITY FULLERTON

<u>Def: First (α)</u> Consider every string derivable from α by a left most derivation.
If α => β where β begins with some terminal, then that terminal is in First (α).

<u>Computation of First (α)</u>

1) if α begins with a terminal t , then First(α) = t
2) If α begins with a nonterminal A, then First (α) includes First(A) – ε
   - **and if A => ε then include First (γ) where α = Aγ**
   - **and if α => ε, then First(α) includes** ε

3) First (ε) = ε

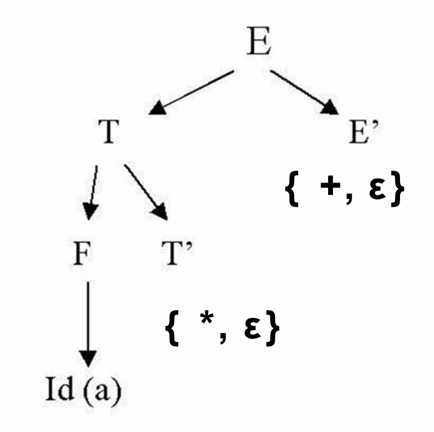
CALIFORNIA STATE UNIVERSITY FULLERTON

Example

s E ->
TE’
E ‘ -> + TE’ | ε
 T -> FT’
T’ -> * FT’ | ε
 F -> id | (E)

First (F) = First
(id) U First
((E)) = { id, (
}
First (T’) =
First (*FT’) |
First (ε) = { *,
ε }
First (T) = First
(FT’) = First
(F) -   ε = {id,
(}
First (E’) =
First (+TE’) U

However, there is a problem using just the First sets.
Consider the previous arithmetic grammar example and a string “ a +b”



Next token = + but is not in First (T’) = { *, ε}
Does that mean it is wrong?
NO, because T’ => ε

In that case, we have to consider what can follow after T’ = { + } => acceptable tokens

So, because of the ε, we need to consider also
Follow (N = nonterminal) = { terminals that can follow right after N}

Follow (A):

Definition: Follow (A) is the set of all terminal symbols that can come right after A in any sentential form of L(G).

If A comes at the end of, the Follow(A) includes "$" = end of file marker

Computation:

IF A is the starting symbol, then include $ in Follow(A)

For all occurrences of A on the RHS of productions do as follows:

Let Q -> α A β (means α before A and β after A), then

If β begins with a terminal t, then t is in Follow(A)

If β begins with a nonterminal , then include First (β) – ε If β => ε or β = ε, then include Follow (Q) in Follow(A)

(we ignore the case Q = A, e.g, A -> α A β)**

CALIFORNIA STATE UNIVERSITY FULLERTON

Example

s E ->
TE'
E' -> + TE' | ε
T -> FT'
T' -> * FT' | ε
F -> id | (E)

Follow (E) =
{$} U

{) } = { $,)}
Follow (E') =
Follow (E) U
Follow (E')
(ignore) =
{ $,)}
Follow (T) =
First (E') – ε U
Follow (E) U
Follow (E') =

Predictive RDP with First and Follow Sets :

```
Procedure E ()
{
If token in First (E) then
      T();
E'();
else error-message (token
in First
      of (E) expected)
}
```

```
Procedure T()
{
     If token in First (T) then
          F();
     T'();
 else error-message (……)
}
```

```
Procedure E' ()
{
   If token = + then
      Lexer();
      T();
      E'();
   else if token not in Follow E'  then

         error-message (……)
}
```

```
Procedure T'()
{  If token = '*' then
        Lexer();
   F();
   T'();
   else if token NOT in folllow (T') then
         error-message (……..)
}
```

```
Procedure F() {
 …… same ….}
```

Productions:

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> * FT' | ε
F -> (E) | id

CALIFORNIA STATE UNIVERSITY FULLERTON

C) Table Driven Predictive Parser

Consist of 3 components:
Parsing table, Stack, Program Driver

Table : to generate a table with all t (columns) and NT (rows)
Ex. For expression grammar

Example:

E -> TE'
E' -> +TE' |
ε T -> FT'
T' -> * FT' |
ε F ->
(E) | id

	id	+	*	(	)	$
E	TE'			TE'		
E'		+TE'			ε	ε
T	FT'			FT'		
T'		ε	*FT'		ε	ε
F	id			(E)		

CALIFORNIA STATE UNIVERSITY FULLERTON

•Stack => well known with pop(), push(), etc

•Driver:

```
Push $ onto the stack
Put end-of-file marker ($) at the end of the input string
 Push (Starting Symbol) on to the stack
While stack not empty do
        let t = TOS symbol and i=incoming token
        if t = terminal symbol then
              if t=i then
                 pop(t); lexer()
              else error-message (…._
        else begin
              if Table [t, i] has entry then
                  pop(t);
                  push Table[t, i] in reverse order
              else error
              end
  endwhile
```

	id	+	*	(	)	$
E	TE’			TE’		
E’		+TE’			ε	ε
T	FT’			FT’		
T’		ε	*FT’		ε	ε
F	id			(E)		

Ex: String   b + c

Stack	***Input***	***Action***
$ E	**b+c$**	**pop(E), Push (E’, T)**
$E’T	**b+c$**	**pop(T), push(T’, F)**
$E’T’F	**b+c$**	**pop(F), push (id);**
$E’T’ id	**b+c$**	**pop(id), lexer();**
$E’T’	**+c$**	**pop(T’); push (ε)**
$E’	**+c$**	**pop(E’); push (E’, T, +)**
$E’T+	**+c$**	**pop(+), lexer()**
$E’T	**c$**	**pop(T), push (T’,F)**
$E’T’F	**c$**	**pop(F); push (id);**
$E’T’id	**c$**	**pop(id), lexer()**
$E’T’	**$**	**pop(T’), push (ε)**
$E’	**$**	**pop(E’), push (ε)**
$	**$**	**Stack empty**

Q: How do we construct such a table?

```
For each Non-terminal N do
{
        Let N ->  β a typical production
         Computer First (β);
         Each terminal t in First (β) except ε
                          do Table [N, t] = β
 If First (β) has ε then
            For each terminal t in Follow (N) do
                  Table [N, t] = ε
}
```

Example:

E -> TE’
E’ -> +TE’ |
ε T -> FT’
T’ -> * FT’ | ε
F -> (E) | id

Computer First Sets:
First (F) = {(, id}
First (T’) = (*, ε)
First (T) = First (F) – ε = {(, id}
First(E’) = {+, ε}
First (E) = First(T) – ε = {(, id}

If a First set contains ε, then compute Follow sets

Follow (T’) = { +, $,)}
Follow (E’) = {), $}

	id	+	*	(	)	$
E	TE’			TE’		
E’		+TE’			ε	ε
T	FT’			FT’		
T’		ε	*FT’		ε	ε
F	id			(E)		

Conflict:
In a table driven predictive parser, a conflict occurs if a cell Table [N, t] has More than one entry.

Ex. "Dangling else" problem
1) S -> if C then S E | a | b
2) C -> x | y
3) E -> else S |

Let's try to construct the table in particular for E
First (E) = {else, ε}

Since it contains ε
We need to computer Follow (E) = Follow (S) =
{ $} U First (E) – ε U Follow(E)
= {S} U { else} = {$, else}

							else	$
S								
C								
E							else S ε	ε

Two entries in Table [E, else] => conflict

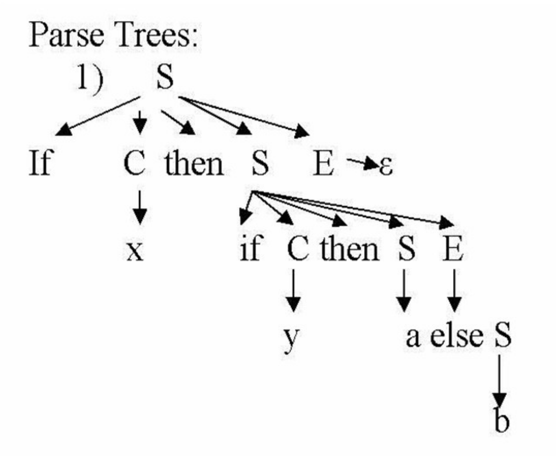
CALIFORNIA STATE UNIVERSITY FULLERTON

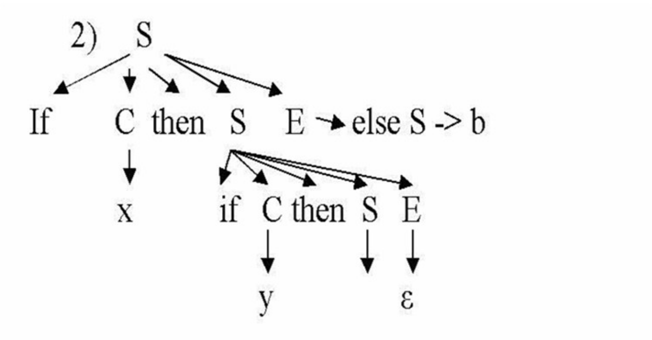
Why Conflict?

a sample sentence:

If x then
If y then a
else b

(where does this else belong????)





Grammar is Ambiguous

=> If a grammar is ambiguous, it will create a conflict

END