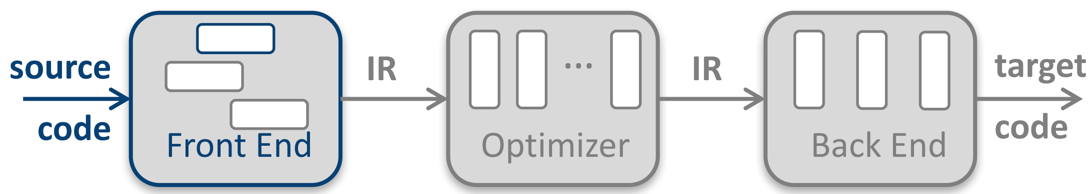
COMP 412
FALL 2017

Syntax Analysis, III

Comp 412



Chapter 3 in EaC2e

The Classic Expression Grammar

Review

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

Classic Expression Grammar, Left-recursive Version

This left-recursive expression grammar encodes standard algebraic precedence and left-associativity (which corresponds to left-to-right evaluation).

We refer to this grammar as either the classic expression grammar or the canonical expression grammar

➡ Both have the same acronym, **CEG**

Top-down Parsing

Review

The Algorithm

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

Construct the root node of the parse tree

Repeat until lower fringe of the parse tree matches the input string

1. *At a node labeled with **NT** A**, select a produc**tion 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 $\in$ NT)*

The key is selecting the right production in step 1

- *That choice should be guided by the input string*

Top-down parsing with the **CEG**

Review

With deterministic choices, the parse can expand indefinitely

Rule	*Sentential Form*	*Input*
—	*Goal*	↑x - 2 * y
0	*Expr*	↑x - 2 * y
1	*Expr +Term*	↑x - 2 * y
1	*Expr + Term +Term*	↑x - 2 * y
1	*Expr + Term +Term + Term*	↑x - 2 * y
1	*… and so on ….*	↑x - 2 * y

Consumes no input!

This expansion doesn't terminate

- 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

Recursion in the **CEG**

Review

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

Classic Expression Grammar, Left-recursive Version

The problems with unbounded expansion arise from left-recursion in the **CEG**

- **LHS** symbol cannot derive, in one or more steps, a sentential form that starts with the **LHS**
- Left recursion is incompatible with top-down leftmost derivations †
- A leftmost derivation matches the scanner's left-to-right scan

We need a technique to transform left recursion into right recursion (and, possible, the reverse)

†Similarly, right recursion is incompatible with rightmost derivaions

Eliminating Left Recursion

Review

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$$\begin{aligned} \textit{Fee} &\rightarrow \textit{Fee}\ \alpha \\ &\mid\ \beta \end{aligned}$$

where neither α nor β start with *Fee*

Language is β followed by 0 or more α's

We can rewrite this fragment as

$$\begin{aligned} \textit{Fee} &\rightarrow \beta\ \textit{Fie} \\ \textit{Fie} &\rightarrow \alpha\ \textit{Fie} \\ &\mid\ \varepsilon \end{aligned}$$

where *Fie* is a new non-terminal

The new grammar defines the same language as the old grammar, using only right recursion.

Added a reference to the empty string

New Idea: the ε production

Eliminating Left Recursion

Review

The expression grammar contains two cases of left recursion

Expr	$\rightarrow$	*Expr + Term*	*Term*	$\rightarrow$	*Term * Factor*
	|	*Expr - Term*		|	*Term * Factor*
	|	*Term*		|	*Factor*

Applying the transformation yields

Expr	$\rightarrow$	*Term Expr’*	*Term*	$\rightarrow$	*Factor Term’*
Expr’	$\rightarrow$	*+ Term Expr’*	*Term’*	$\rightarrow$	** Factor Term’*
	|	*- Term Expr’*		|	*/ Factor Term’*
	|	ε		|	ε

These fragments use only right recursion

Eliminating Left Recursion

Review

Substituting them back into the grammar yields

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

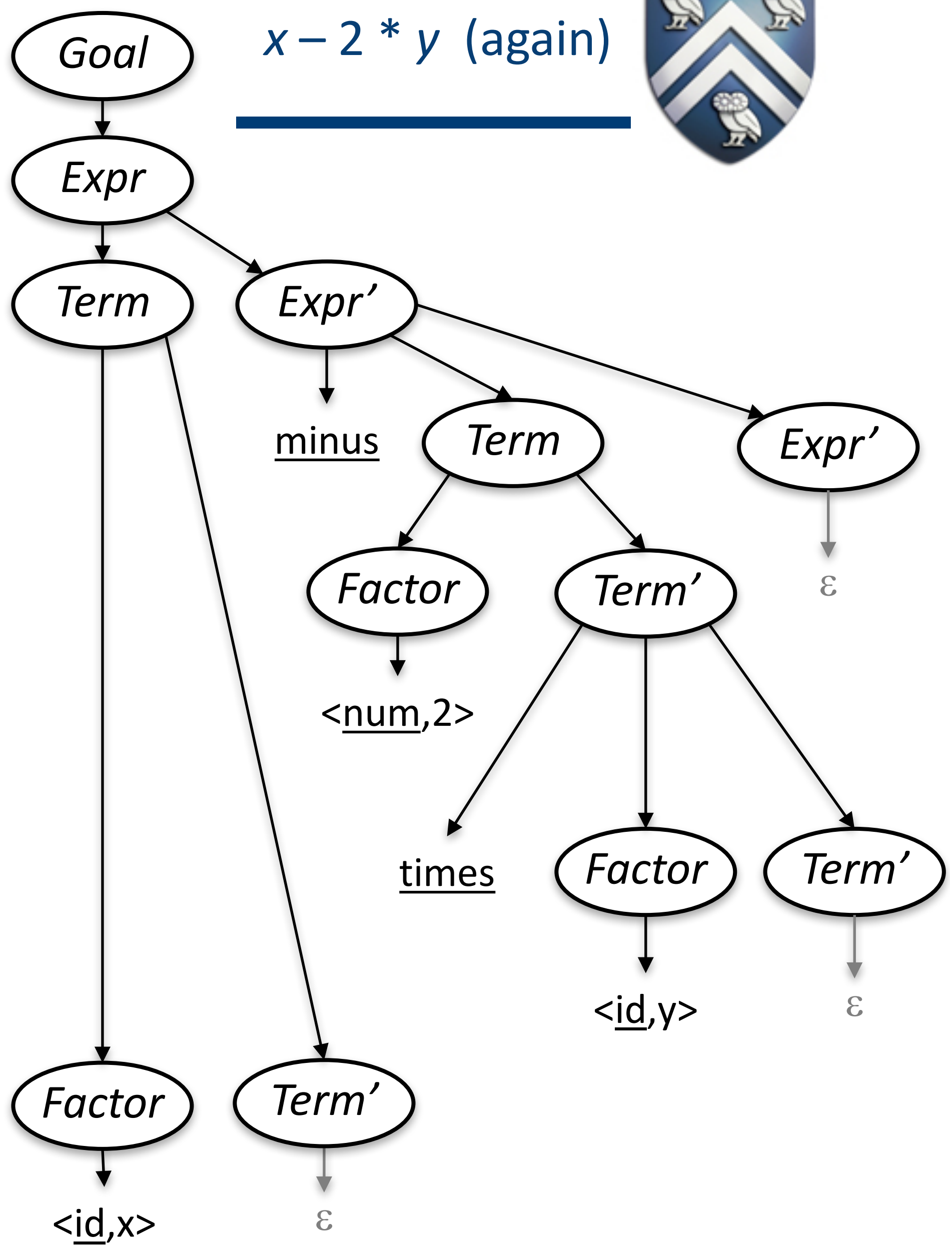
Right-recursive expression grammar

- This grammar is correct, if somewhat counter-intuitive.
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.
- It is left associative, as was the original

Parsing with **RR CEG**

$x - 2 * y$ (again)

Rule	*Sentential Form*
—	*Goal*
0	*Expr*
1	*Term Expr’*
5	*Factor Term’ Expr’*
11	<id,x> *Term’ Expr’*
8	<id,x> *Expr’*
3	<id,x> - *Term Expr’*
5	<id,x> - *Factor Term’ Expr’*
10	<id,x> - <num,2> *Term’ Expr’*
6	<id,x> - <num,2> * *Factor Term’ Expr’*
11	<id,x> - <num,2> * <id,y> *Term’ Expr’*
8	<id,x> - <num,2> * <id,y> *Expr’*
4	<id,x> - <num,2> * <id,y>



Eliminating Left Recursion

*"**LHS** symbol cannot derive, in one or **more** steps, a sentential form that starts with the **LHS**"*

The two-production transformation eliminates immediate left recursion

What about more general, indirect left recursion ?

The general algorithm to eliminate left recursion:

arrange the NTs into some order A_1, A_2, …, A_n
for $i \leftarrow 2$ to n
  for $s \leftarrow 1$ to $i - 1$ {
    replace each production $A_i \rightarrow A_s\gamma$ with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
      where $A_s \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current productions for A_s
    }
  eliminate any immediate left recursion on A_i using the direct transformation

The algorithm assumes that the initial grammar has no cycles ($A_i \Rightarrow^+ A_i$), and no epsilon productions

EaC2e shows the *i* loop running from 1 to n, so 1st iteration of inner loop never executes.

Eliminating Left Recursion

How does this algorithm work?

1. Impose arbitrary order on the non-terminals
2. Outer loop cycles through **NT** in order
3. Inner loop ensures that a production expanding A_i cannot directly derive a non-terminal A_s at the start of its **RHS**, for $s < i$
4. Last step in outer loop converts any direct recursion on A_i to right recursion using the transformation showed earlier
5. New non-terminals are added at the end of the order & have no left recursion; they will have right recursion on themselves

At the start of the i^{th} outer loop iteration

For all $k < i$, no production that expands A_k begins with a non-terminal A_s, for $s < k$

Eliminating Indirect Left Recursion

0	$Start_1$	$\rightarrow$	A_2
1	A_2	$\rightarrow$	B_3 <u>a</u>
2		|	<u>a</u>
3	B_3	$\rightarrow$	A_2 <u>b</u>

Example Grammar

- This grammar generates **<u>a</u> (<u>ba</u>)***
- Subscripts indicate the imposed order
- Indirect left recursion is $A \rightarrow B \rightarrow A$

arrange the NTs into some order $A_1, A_2, \ldots, A_n$
for $i \leftarrow 2$ *to* n
    for $s \leftarrow 1$ *to* $i - 1$ {
        replace each production $A_i \rightarrow A_s\gamma$ *with* $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
            where $A_s \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ *are all the current productions for* A_s
        }
    eliminate any immediate left recursion on A_i *using the direct transformation*

Eliminating Indirect Recursion

General algorithm is a diagonalization, similar to LU decomposition

	NT_1	NT_2	NT_3
NT_1	0	**1**	0
NT_2	0	0	**1**
NT_3	0	**1**	0
Original Grammar			

0	$Start_1$	$\rightarrow$	A_2
1	A_2	$\rightarrow$	B_3 a
2		|	a
3	B_3	$\rightarrow$	A_2 b

- If [i, j] element is *k*, then NT_i derives NT_j in a the k^{th} position of its **RHS**
- Diagonal elements represent self-recursion
- Upper triangle represents forward references in the order
- Lower triangle represents backward references in the order
  - *Inner loop systematically zeroes the lower triangle*
  - *Last step of outer loop eliminates ones on the diagonal*

Eliminating Indirect Left Recursion

Applying the Algorithm

	s = 1	*2*	*3*
***i* = 1**	*forward*	*forward*	*forward*
2	no action	*forward*	*forward*
3	no action	*see below*	*forward*

Algorithm iterates through the possible backward references.

- Handles indirect left recursion with forward substitution of **RHS** for **LHS**
- Handles direct left recursion with the transformation

For B_3 (i = 3) and A_2 (s = 2):

First, it rewrites rule 3 with

$$B_3 \rightarrow B_3 \ \underline{a} \ \underline{b}$$
$$\quad | \ \underline{a} \ \underline{b}$$

Next, it uses the rule for immediate left recursion to rewrite this pair of rules as:

$$B_3 \rightarrow \underline{a} \ \underline{b} \ C$$
$$C \rightarrow \underline{a} \ \underline{b} \ C$$
$$\quad | \ \varepsilon$$

Eliminating Indirect Left Recursion

Before and After

0	$Start_1$	$\rightarrow$	A_2
1	A_2	$\rightarrow$	B_3 $\underline{\text{a}}$
2		|	$\underline{\text{a}}$
3	B_3	$\rightarrow$	A_2 $\underline{\text{b}}$

Original Grammar

0	$Start_1$	$\rightarrow$	A_2
1	A_2	$\rightarrow$	B_3 $\underline{\text{a}}$
2		|	$\underline{\text{a}}$
3	B_3	$\rightarrow$	$\underline{\text{a}}$ $\underline{\text{b}}$ C
4	C	$\rightarrow$	$\underline{\text{a}}$ $\underline{\text{b}}$ C
5		|	ε

Transformed Grammar

- The transformed grammar generates $\underline{\text{a}}\ (\underline{\text{ba}})^*$

Eliminating Indirect Recursion

General algorithm is a diagonalization, similar to LU decomposition

	NT_1	NT_2	NT_3
NT_1	0	**1**	0
NT_2	0	0	**1**
NT_3	0	1	0

Original Grammar

0	$Start_1$	$\rightarrow$	A_2
1	A_2	$\rightarrow$	B_3 <u>a</u>
2		|	<u>a</u>
3	B_3	$\rightarrow$	A_2 <u>b</u>

	NT_1	NT_2	NT_3	NT_4
NT_1	0	**1**	0	0
NT_2	0	0	**1**	0
NT_3	0	0	0	**3**
NT_4	0	0	0	**3**

Transformed Grammar

0	$Start_1$	$\rightarrow$	A_2
1	A_2	$\rightarrow$	B_3 <u>a</u>
2		|	<u>a</u>
3	B_3	$\rightarrow$	<u>a</u> <u>b</u> C
4	C	$\rightarrow$	<u>a</u> <u>b</u> C
5		|	ε

Eliminating Indirect Left Recursion

Before and After

0	$Start_1$	$\rightarrow$	A_2
1	A_2	$\rightarrow$	B_3 $\underline{a}$
2		|	$\underline{a}$
3	B_3	$\rightarrow$	A_2 $\underline{b}$

Original Grammar

0	$Start_1$	$\rightarrow$	A_2
1	A_2	$\rightarrow$	B_3 $\underline{a}$
2		|	$\underline{a}$
3	B_3	$\rightarrow$	$\underline{a}$ $\underline{b}$ C
4	C	$\rightarrow$	$\underline{a}$ $\underline{b}$ C
5		|	ε

Transformed Grammar

- The transformed grammar generates $\underline{a}$ ($\underline{ba}$)*
- The transformed grammar is still not … able.

What does "predictively parsable" even mean?

Predictive Parsing

If a top-down parser picks the "wrong" production, it may need to backtrack. The alternative is to *look ahead in the input & use context to pick the correct production.*

How much lookahead is needed for a context-free grammar?

- In general, an arbitrarily large amount
- Use the Cocke-Younger, Kasami algorithm or Earley's algorithm

Fortunately,

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are *LL(1)* and *LR(1)* grammars

We will focus, for now, on LL(1) grammars & predictive parsing

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, *the parser should be able to choose between* α & β

FIRST sets

For some **RHS** $\alpha \in G$, 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

We will defer the problem of how to compute **FIRST** sets for the moment and assume that they are given.

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α & β

FIRST sets

For some **RHS** $\alpha \in G$, 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

The Predictive, or LL(1), Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

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

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

This rule is intuitive. Unfortunately, it is **not correct**, because it does not handle ε rules. See the next slide

Predictive Parsing

What about ε-productions?

$\Rightarrow$ They complicate the definition of the predictive, or **LL(1)** property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ FIRST(α), then we need to ensure that FIRST(β) is disjoint from FOLLOW(A), too, where

FOLLOW(A) *is the set of terminal symbols that can appear immediately after A in some sentential form*

Define **FIRST**$^+$($A \rightarrow \alpha$) as

Note that **FIRST**$^+$ is defined over productions, not **NT**s

- **FIRST**(α) $\cup$ **FOLLOW**(A), if $\varepsilon \in$ **FIRST**(α)
- **FIRST**(α), otherwise

Then, a grammar is **LL(1)** *iff* $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies that

FIRST$^+$($A \rightarrow \alpha$) $\cap$ **FIRST**$^+$($A \rightarrow \beta$) = $\varnothing$

The **LL(1)** Property

Our transformed grammar for a (ba)* fails the test

0	$Start_1$	$\rightarrow$	A_2
1	A_2	$\rightarrow$	B_3 a
2		|	a
3	B_3	$\rightarrow$	a b C
4	C	$\rightarrow$	a b C
5		|	ε

Transformed Grammar

Derivations for A begin with a:

$A \rightarrow B$ a | a

$B \rightarrow$ a

FIRST$^+$($A \rightarrow B$ a) = { a }

FIRST$^+$($A \rightarrow$ a) = { a }

These sets are identical, rather than disjoint. So, left recursion elimination did not produce a grammar with the LL(1) condition — a grammar that would be predictively parsable.

The **LL(1)** Property

Can we write an LL(1) grammar for a (ba)* ?

0	$Start_1$	$\rightarrow$	A_2
1	A_2	$\rightarrow$	B_3 a
2		|	a
3	B_3	$\rightarrow$	a b C
4	C	$\rightarrow$	a b C
5		|	ε

Transformed Grammar

0	$Start$	$\rightarrow$	a B
1	B	$\rightarrow$	b a B
2		|	ε

LL(1) Grammar for a (ba)*

What If My Grammar Is Still Not LL(1) ?

Can we transform a non-LL(1) grammar into an LL(1) grammar?

- In general, the answer is no
- In some cases, however, the answer is yes

Assume a grammar G with productions $A \rightarrow \alpha\,\beta_1$ and $A \rightarrow \alpha\,\beta_2$

- If α derives anything other than ε, then

$$\textsc{First}^+(A \rightarrow \alpha\,\beta_1) \cap \textsc{First}^+(A \rightarrow \alpha\,\beta_2) \neq \varnothing$$

- And the grammar is not LL(1)

If we pull the common prefix, α, into a separate production, we **may** make the grammar LL(1).

$$A \rightarrow \alpha\, A', \;\; A' \rightarrow \beta_1, \;\; \text{and} \;\; A' \rightarrow \beta_2$$

Now, if $\textsc{First}^+(A' \rightarrow \beta_1) \cap \textsc{First}^+(A' \rightarrow \beta_2) = \varnothing$, G may be LL(1)

What If My Grammar Is Not LL(1) ?

Left Factoring

*For each **NT** A*

find the longest prefix α common to 2 or more alternatives for A

if $\alpha \neq \varepsilon$ then

replace all of the productions

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

with

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$$

*Repeat until no **NT** has alternative rhs' with a common prefix*

This transformation makes some grammars into **LL(1)** grammars

There are languages for which no **LL(1)** grammar exists

Left Factoring Example

Consider a short grammar for subscripted identifiers

0	*Factor*	$\rightarrow$	name
1		|	name [*ArgList*]
2		|	name (*ArgList*)
3	*ArgList*	|	*Expr MoreArgs*
4	*MoreArgs*	$\rightarrow$	, *Expr MoreArgs*
5		|	ε

Common prefix is name. Introduce a new production

Factor $\rightarrow$ name *Arguments*

And expand *Arguments* into the three alternatives.

To choose between expanding by 0, 1, or 2, a top-down parser must look beyond the name to see the next word.

$$\text{First}^+(0) = \text{First}^+(1) = \text{First}^+(2) = \{\ \underline{\text{name}}\ \}$$

Left factoring productions 0, 1, & 2 fixes this problem.

Left factoring cannot fix all non-**LL(1)** grammars.

Left Factoring Example

After left factoring:

0	*Factor*	→	name *Args*
1	*Args*	→	[*ArgList*]
2		|	(*ArgList*)
3		|	ε
4	*ArgList*	|	*Expr MoreArgs*
5	*MoreArgs*	→	, *Expr MoreArgs*
6		|	ε

Left factoring can transform some non-**LL(1)** grammars into **LL(1)** grammars.

There are languages for which no **LL(1)** grammar exists.

See exercise 3.12 in EaC2e.

Clearly, $\textbf{FIRST}^{+}(1)$ & $\textbf{FIRST}^{+}(2)$ are disjoint.

If neither (or [can follow *Factor*, then $\textbf{FIRST}^{+}(3)$ will be, too.

This tiny grammar now has the **LL(1)** property

Predictive Parsing

Given a grammar that has the LL(1) property

- We can write a simple routine to recognize an instance of each **LHS**
- Code is patterned, simple, & fast

Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

$$\text{FIRST}^+(A\rightarrow\beta_i) \cap \text{FIRST}^+(A\rightarrow\beta_j) = \varnothing \text{ if } i \neq j$$

/* find an A */
if (current_word $\in$ FIRST$^+(A\rightarrow\beta_1)$)
   find a β_1 and return true
else if (current_word $\in$ FIRST$^+(A\rightarrow\beta_2)$)
   find a β_2 and return true
else if (current_word $\in$ FIRST$^+(A\rightarrow\beta_3)$)
   find a β_3 and return true
else
   report an error and return false

Grammars that have the **LL(1)** property are called **predictive grammars** because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the **LL(1)** property are called **predictive parsers**.

One kind of predictive parser is the **recursive descent** parser.

Of course, there is more detail to "find a β_i" — typically a recursive call to another small routine (see pp. 108--111 in EaC2e)

Recursive Descent Parsing

Recall the expression grammar, after transformation

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

This grammar leads to a parser that has six **mutually recursive** routines:

1. *Goal*
2. *Expr*
3. *EPrime*
4. *Term*
5. *TPrime*
6. *Factor*

Each routine recognizes an ***rhs*** for that **NT**.

The term **descent** refers to the direction in which the parse tree is built.

This grammar has the LL(1) property

Recursive Descent Parsing (Procedural)

A couple of routines from the expression parser

```
Goal( )
    token ← next_token( );
    if (Expr( ) = true & token = EOF)
        then next compilation step;
        else
            report syntax error;
            return false;

Expr( )
  if (Term( ) = false)
     then return false;
     else return Eprime( );
```

looking for <u>number</u>, <u>identifier</u>, or <u>(</u>, found token instead, or failed to find *Expr* or <u>)</u> after <u>(</u>

```
Factor( )
  if (token = number) then
     token ← next_token( );
     return true;
  else if (token = identifier) then
     token ← next_token( );
     return true;
  else if (token = lparen)
     token ← next_token( );
     if (Expr( ) = true & token = rparen) then
        token ← next_token( );
        return true;
  // fall out of if statement
  report syntax error;
     return false;
```

EPrime, Term, & TPrime follow the same basic lines (Figure 3.10, EaC2e)

Recursive Descent

Page 111 in EaC2e sketches a recursive descent parser for the RR CEG.

- One routine per **NT**
- Check each **RHS** by checking each symbol
- Includes ε-productions

Your lab 2 parsers are not much more complex than the example.

```
Main()
  /* Goal → Expr */
  word ← NextWord();
  if (Expr())
    then if (word = eof)
      then report success;
      else Fail();

Fail()
  report syntax error;
  attempt error recovery or exit;

Expr()
  /* Expr → Term Expr′ */
  if (Term())
    then return EPrime();
    else Fail();

EPrime()
  /* Expr′ → + Term Expr′ */
  /* Expr′ → - Term Expr′ */
  if (word = + or word = -)
    then begin;
      word ← NextWord();
      if (Term())
        then return EPrime();
        else Fail();
    end;
  else if (word = ) or word = eof)
    /* Expr′ → ϵ */
    then return true;
    else Fail();

Term()
  /* Term → Factor Term′ */
  if (Factor())
    then return TPrime();
    else Fail();

TPrime()
  /* Term′ → × Factor Term′ */
  /* Term′ → ÷ Factor Term′ */
  if (word = × or word = ÷)
    then begin;
      word ← NextWord();
      if (Factor())
        then return TPrime();
        else Fail();
    end;
  else if (word = + or word = - or
          word = ) or word = eof)
    /* Term′ → ϵ */
    then return true;
    else Fail();

Factor()
  /* Factor → ( Expr ) */
  if (word = ( ) then begin;
    word ← NextWord();
    if (not Expr())
      then Fail();
    if (word ≠ ) )
      then Fail();
    word ← NextWord();
    return true;
  end;
  /* Factor → num */
  /* Factor → name */
  else if (word = num or
          word = name)
    then begin;
      word ← NextWord();
      return true;
    end;
  else Fail();
```

Top-Down Recursive Descent Parser

At this point, you almost have enough information to build a top-down recursive-descent parser

- Need a right-recursive grammar that meets the **LL(1)** condition
  - Can use left-factoring to eliminate common prefixes
  - Can transform direct left recursion into right recursion
  - Need a general algorithm to handle indirect left recursion
- Need algorithms to construct **FIRST** and **FOLLOW**

Next Class

- Algorithms for **FIRST** and **FOLLOW**
- Algorithm for generalized elimination of left recursion
- An **LL(1)** skeleton parser, table, and table-construction algorithm