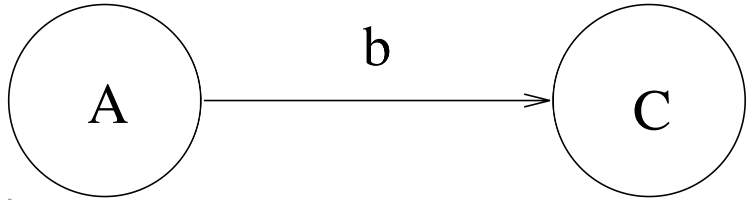
Beyond finite-state machines

For a rule of the form

$$\textbf{A} \rightarrow \textbf{b C}$$

we developed a finite-state mechanism of the form



After arrival at C, there is no need to remember how we got there.

Now, with a rule such as

$$\textbf{F} \rightarrow \textbf{(E)}$$

we cannot just arrive at an E and forget that we need exactly one closing parenthesis for each opening one that got us there.

Instead of "going to" a state E based on consuming an opening parenthesis, suppose we called a procedure E to consume all input ultimately derived from the nonterminal:

Procedure $F()$

  call $Expect(OpenParen)$
  call $E()$
  call $Expect(CloseParen)$

end

This style of parser construction is called *recursive descent*. **The procedure associated with each nonterminal is responsible for directing the parse through the right-hand side of the appropriate production.**

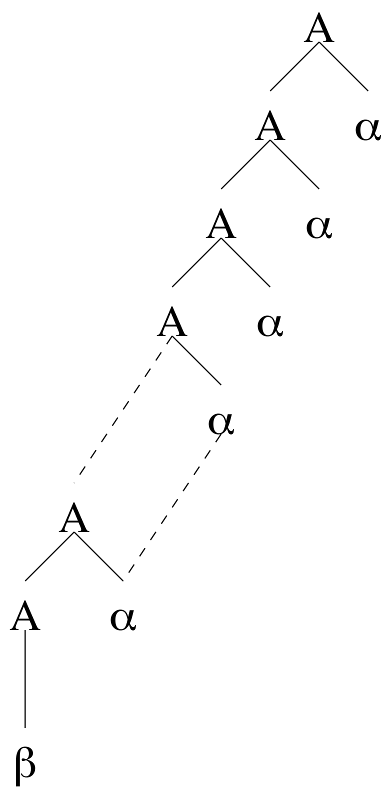
---

1. What about rules that are left-recursive?
2. What happens if there is more than one rule associated with a nonterminal?

Eliminating left recursion – grammar transformation

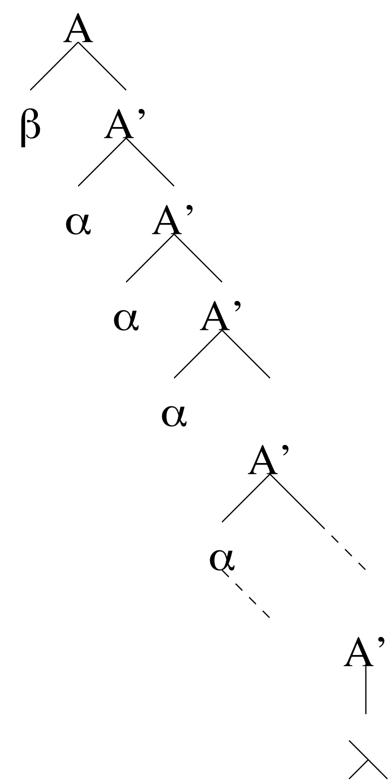
Original

$$\begin{aligned} \mathbf{A} &\rightarrow \mathbf{A}\,\alpha \\ &\mid\ \beta \end{aligned}$$



Transformed

$$\begin{aligned} \mathbf{A} &\rightarrow \beta\, A' \\ A' &\rightarrow \alpha\, A' \ \mid\ \lambda \end{aligned}$$

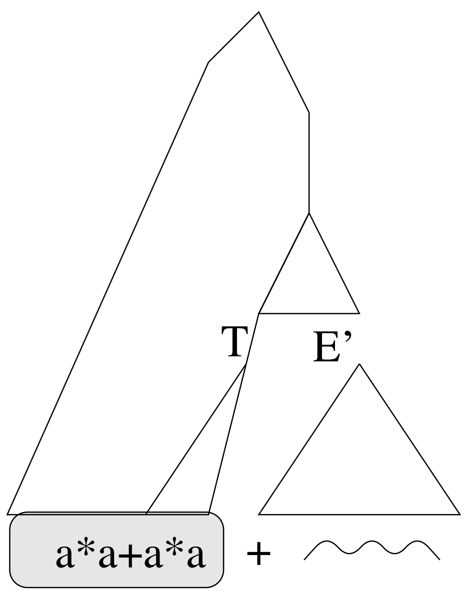


The two grammars generate the same language, but the one on the right generates the β first, and then a string of αs, using a rule that is *right* recursive instead of left recursive.

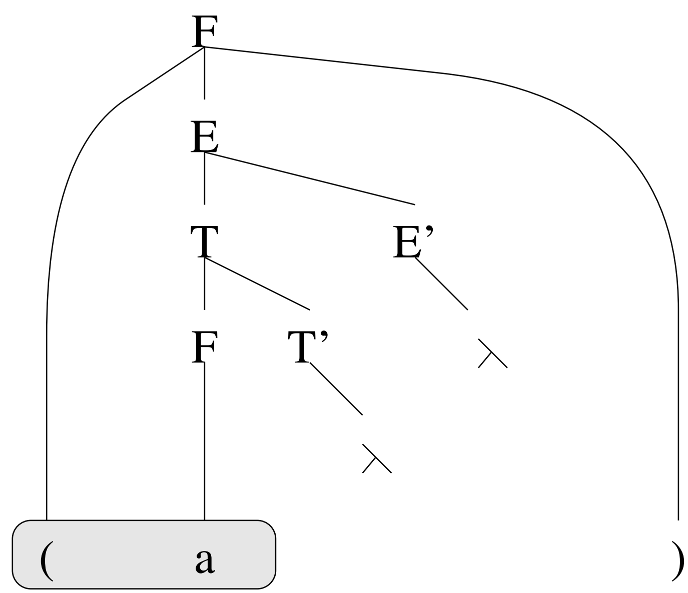
The transformed expression grammar

$$
\begin{array}{lcl}
\textbf{E} & \rightarrow & \textbf{T}\ E' \\
\hline
E' & \rightarrow & \textbf{+ T}\ E' \\
E' & \rightarrow & \textbf{- T}\ E' \\
 & | & \lambda \\
\hline
\textbf{T} & \rightarrow & \textbf{F}\ T' \\
T' & \rightarrow & *\ \textbf{F}\ T' \\
T' & \rightarrow & \textbf{/ F}\ T' \\
 & | & \lambda \\
\textbf{F} & \rightarrow & \textbf{(E)} \\
 & | & \textbf{a}
\end{array}
$$

Which rule to choose?



And what about λ?



First sets

$$First(\alpha) = \begin{cases} \{\,\alpha\,\} & \textbf{if } \alpha \in \Sigma \\ \cup_{(\alpha \to \omega_i) \in P} First(\omega_i) & \textbf{if } \alpha \in V \\ \{\,\lambda\,\} & \textbf{if } \alpha = \lambda \end{cases}$$

$$First(\alpha_1 \ldots \alpha_L) = \bigcup_{j \,\mid\, \forall_{k=1}^{j-1} (\lambda \in First(\alpha_k))} First(\alpha_j)$$

A	$\rightarrow$	**B C**
	$\mid$	**E F G H**
	$\mid$	**H**
B	$\rightarrow$	**b**
C	$\rightarrow$	λ
	$\mid$	**c**
E	$\rightarrow$	λ
	$\mid$	**e**
F	$\rightarrow$	**C E**
G	$\rightarrow$	**g**
H	$\rightarrow$	λ
	$\mid$	**h**

ω	**First(ω)**
H	$\{\,h, \lambda\,\}$
G	$\{\,g\,\}$
C	$\{\,c, \lambda\,\}$
B	$\{\,b\,\}$
E	$\{\,e, \lambda\,\}$
F	$\{\,c, e, \lambda\,\}$
A	$\{\,b, e, c, g, h, \lambda\,\}$
BC	$\{\,b\,\}$
EFGH	$\{\,e, c, g\,\}$

Follow sets

1. **Initially set** $Follow(N) = \emptyset, \forall\ N \in V$.
2. **Given production** $A \rightarrow \alpha B \beta$**, set**

$$Follow(B) = Follow(B) \cup (First(\beta) - \{\lambda\})$$

3. **Given production** $A \rightarrow \alpha B \beta$**, where** $\lambda \in First(\beta)$**, set**

$$Follow(B) = Follow(B) \cup Follow(A)$$

A	$\rightarrow$	**B C**
	$\mid$	**E F G H**
	$\mid$	**H**
B	$\rightarrow$	**b**
C	$\rightarrow$	λ
	$\mid$	**c**
E	$\rightarrow$	λ
	$\mid$	**e**
F	$\rightarrow$	**C E**
G	$\rightarrow$	**g**
H	$\rightarrow$	λ
	$\mid$	**h**

N	**Follow(N)**		
A			$\{\ \}$
B	$First(C) \cup Follow(A)$	=	$\{c\}$
F	$First(G)$	=	$\{g\}$
C	$Follow(A) \cup First(E)$		
	$\cup Follow(F)$	=	$\{e, g\}$
E	$First(F) \cup First(G)$	=	$\{c, e, g\}$
G	$First(H) \cup Follow(A)$	=	$\{h\}$
H	$Follow(A)$	=	$\{\ \}$

Recursive descent parser generation

```
Procedure NonTermN
    if (LookAhead() ∈ First(ω1), where (N → ω1) ∈ P) then
        /* Use ω1 to generate calls to Expect() and other nonterminals */
    else
        if (LookAhead() ∈ Follow(N) and (N → λ) ∈ P) then
            return ()
        else
            /* error */
        fi
    fi
end
```

Recursive descent – Example

$$
\begin{array}{lcl}
\textbf{S} & \rightarrow & \textbf{A C \$} \\
\textbf{C} & \rightarrow & \textbf{c} \\
 & | & \lambda \\
\textbf{A} & \rightarrow & \textbf{a B C d} \\
 & | & \textbf{B Q} \\
 & | & \lambda \\
\textbf{B} & \rightarrow & \textbf{b B} \\
 & | & \textbf{d} \\
\textbf{Q} & \rightarrow & \textbf{q}
\end{array}
$$

	First	**Follow**
S	$\{ a, b, d, c, \$ \}$	$\{ \}$
A	$\{ a, b, d, \lambda \}$	$\{ c, \$ \}$
B	$\{ b, d \}$	$\{ c, d, q \}$
C	$\{ c, \lambda \}$	$\{ d, \$ \}$
Q	$\{ q \}$	$\{ c, \$ \}$

The generated procedures

S	$\rightarrow$	A C \$
C	$\rightarrow$	c
	$\mid$	λ
A	$\rightarrow$	a B C d
	$\mid$	B Q
	$\mid$	λ
B	$\rightarrow$	b B
	$\mid$	d
Q	$\rightarrow$	q

	First	Follow
S	$\{a, b, d, c, \$\}$	$\{\ \}$
A	$\{a, b, d, \lambda\}$	$\{c, \$\}$
B	$\{b, d\}$	$\{c, d, q\}$
C	$\{c, \lambda\}$	$\{d, \$\}$
Q	$\{q\}$	$\{c, \$\}$

```
Procedure S()
    if (LookAhead() ∈ { a, b, d, c, $ }) then
        call A()
        call C()
        call Expect($)
    else
        /* error */
    fi

end

Procedure C()
    if (LookAhead() ∈ { c }) then
        call Expect(c)
    else
        if (Lookahead() ∉ { d, $ }) then
            /* error */
        fi
    fi

end
```

The generated procedures (cont'd)

S	$\rightarrow$	**A C $**
C	$\rightarrow$	**c**
	$\mid$	λ
A	$\rightarrow$	**a B C d**
	$\mid$	**B Q**
	$\mid$	λ
B	$\rightarrow$	**b B**
	$\mid$	**d**
Q	$\rightarrow$	**q**

	First	**Follow**
S	$\{a, b, d, c, \$\}$	$\{\ \}$
A	$\{a, b, d, \lambda\}$	$\{c, \$\}$
B	$\{b, d\}$	$\{c, d, q\}$
C	$\{c, \lambda\}$	$\{d, \$\}$
Q	$\{q\}$	$\{c, \$\}$

Procedure $A()$
    if $(LookAhead() \in \{a\})$ **then**
        call $Expect(a)$
        call $B()$
        call $C()$
        call $Expect(d)$
    else
        if $(LookAhead() \in \{b, d\})$ **then**
            call $B()$
            call $Q()$
        else
            if $(LookAhead() \in \{c, \$\})$ **then**
                return ()
            else
                /* *error* */
            fi
        fi
    fi

end

The generated procedures (cont'd)

S	$\rightarrow$	A C $
C	$\rightarrow$	c
	$\mid$	λ
A	$\rightarrow$	a B C d
	$\mid$	B Q
	$\mid$	λ
B	$\rightarrow$	b B
	$\mid$	d
Q	$\rightarrow$	q

	First	Follow
S	$\{a, b, d, c, \$\}$	$\{\}$
A	$\{a, b, d, \lambda\}$	$\{c, \$\}$
B	$\{b, d\}$	$\{c, d, q\}$
C	$\{c, \lambda\}$	$\{d, \$\}$
Q	$\{q\}$	$\{c, \$\}$

```
Procedure B()
    if (LookAhead() ∈ { b }) then
        call Expect(b)
        call B()
    else
        if (LookAhead() ∈ { d }) then
            call Expect(d)
        else
            /* error */
        fi
    fi
end

Procedure Q()
    if (LookAhead() ∈ { q }) then
        call Expect(q)
    else
        /* error */
    fi
end
```

Recursive descent – expression grammar

E	$\rightarrow$	**T** E'
E'	$\rightarrow$	**+ T** E'
E'	$\rightarrow$	**- T** E'
	$\mid$	λ
T	$\rightarrow$	**F** T'
T'	$\rightarrow$	$*$ **F** T'
T'	$\rightarrow$	**/ F** T'
	$\mid$	λ
F	$\rightarrow$	**(E)**
	$\mid$	**a**

	First	Follow
E	$\{ (, a \}$	$\{), \$ \}$
E'	$\{ +, - \}$	$\{), \$ \}$
T	$\{ (, a \}$	$\{ +, -,), \$ \}$
T'	$\{ *, / \}$	$\{ +, -,), \$ \}$
F	$\{ (, a \}$	$\{ *, /, +, -,), \$ \}$

Procedure E'
    if $(LookAhead(+))$ **then**
        call $Expect(+)$
        call T
        call E'
    else
        if $(LookAhead(-))$ **then**
            call $Expect(-)$
            call T
            call E'
        else
            if $(LookAhead(\$, \text{')'}))$ **then**
                return ()
            else
                call $Error()$
            fi
        fi
    fi
end

Maintaining lookahead

Procedure $main()$
  $LAtok \leftarrow GetNextToken()$
  call $S()$
end

Function $LookAhead()$ **:** $token$
  return $(LAtok)$
end

Procedure $Expect(tok)$
  if $(LAtok = tok)$ **then**
    $LAtok \leftarrow GetNextToken()$
  else
    / error */*
  fi
end

A lookahead of k tokens is maintained by appropriately buffering the input.

Technically, k lookahead is equivalent in power to a single token of lookahead. The proof is constructive: each permutation of k symbols is encoded as a single token.

The $Expect(tok)$ procedure first compares the incoming token against tok, and then advances input into the lookahead buffer.

Recursive descent – correctness and properties

When is our recursive descent parser construction successful? If the grammar involves any left-recursion, then our construction method will create a parser containing an infinite loop. So, we require that the grammar be free of left-recursion.

The grammar transformation technique covered earlier can help eliminate left-recursion.

Also, we require that the parser operate *deterministically*: **actions taken at each step make progress toward completion, so that backtracking is not necessary.**

Thus, given a set of rules for nonterminal N

$$\begin{array}{lcl} N & \rightarrow & \omega_1 \\ & | & \vdots \\ & | & \omega_n \end{array}$$

we require

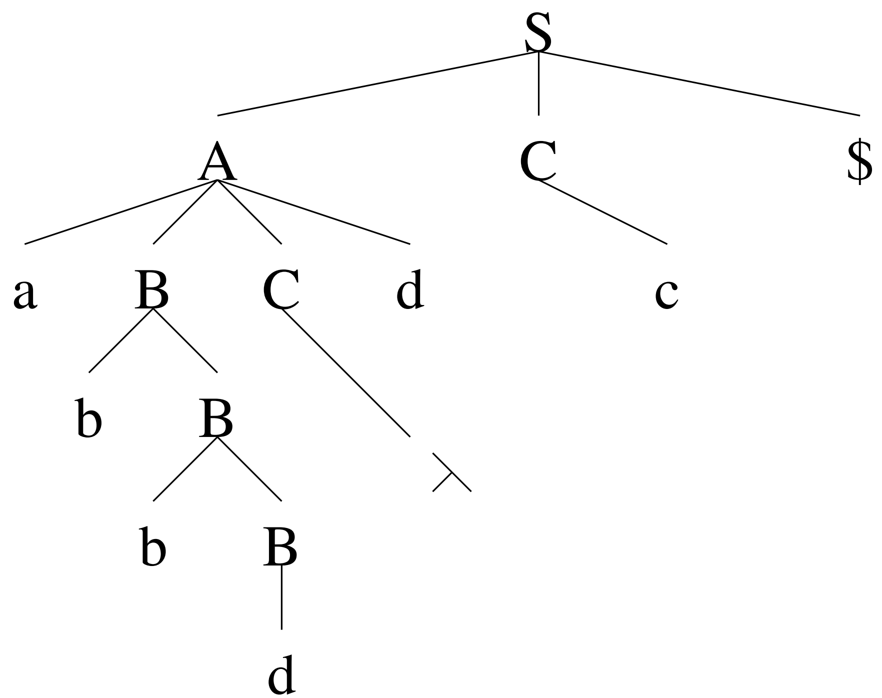
1. $$\bigcap_i First(\omega_i) = \{\ \}$$

2. If $\lambda = \omega_j$, $1 \leq j \leq n$, then we also require

$$\bigcup_i (Follow(N) \cap First(\omega_i)) = \{\ \}$$

Recursive descent and leftmost derivations

Let's examine how our recursive descent parser recognizes the string "abbddc$"



[1]	**S**	$\rightarrow$	**A C $**
[2]	**C**	$\rightarrow$	**c**
[3]		$\mid$	λ
[4]	**A**	$\rightarrow$	**a B C d**
[5]		$\mid$	**B Q**
[6]		$\mid$	λ
[7]	**B**	$\rightarrow$	**b B**
[8]		$\mid$	**d**
[9]	**Q**	$\rightarrow$	**q**

$$
\begin{aligned}
\mathbf{S} &\Rightarrow A\ \mathbf{C}\ \$ \\
&\Rightarrow \boxed{\mathbf{a}\ B\ \mathbf{C}\ \mathbf{d}}\ \mathbf{C}\ \$ \\
&\Rightarrow \mathbf{a}\ \boxed{\mathbf{b}\ B}\ \mathbf{C}\ \mathbf{d}\ \mathbf{C}\ \$ \\
&\Rightarrow \mathbf{a}\ \mathbf{b}\ \boxed{\mathbf{b}\ B}\ \mathbf{C}\ \mathbf{d}\ \mathbf{C}\ \$ \\
&\Rightarrow \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \boxed{\mathbf{d}}\ C\ \mathbf{d}\ \mathbf{C}\ \$ \\
&\Rightarrow \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \mathbf{d}\ \boxed{}\ \mathbf{d}\ C\ \$ \\
&\Rightarrow \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \mathbf{d}\ \mathbf{d}\ \boxed{\mathbf{c}}\ \$
\end{aligned}
$$

The procedure activations trace a left-most derivation of the string. We call this style of parsing *LL*, because it uses a *L*eft-most scan of the input and produces a *L*eft-most derivation.

In fact, the *record* of the parse is simply the order in which the grammar rules are applied: [1] [4] [7] [7] [8] [3] [2]