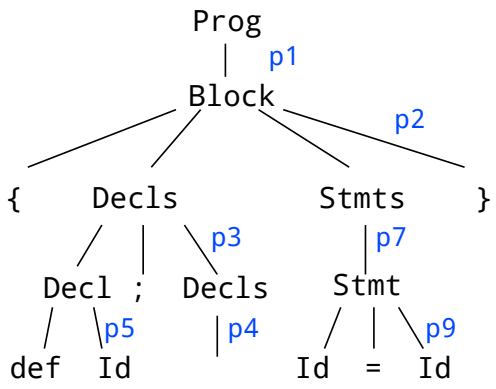
Top-Down and Bottom-Up Parsing

Top-Down and *Bottom-Up* refer to the order in which the derivation tree is constructed.

- Top-Down Parsing:
  - Construction starts at the root and proceeds towards the leaves
  - Predictive Parsers, e.g. LL(1), Recursive-Descent
- Bottom-Up Parsing:
  - Construction starts at the leaves and proceeds towards the root
  - Shift-Reduce Parsers, e.g. LR(1), LALR(1)

```
p1:  Prog  ::= Block
p2:  Block ::= '{' Decls Stmts '}'
p3:  Decls ::= Decl ';' Decls
p4:  Decls ::=
p5:  Decl  ::= 'def' Id
p6:  Stmts ::= Stmts ';' Stmt
p7:  Stmts ::= Stmt
p8:  Stmt  ::= Block
p9:  Stmt  ::= Id '=' Id
```

Grammar



Derivation Tree for `{def Id; Id = Id}`

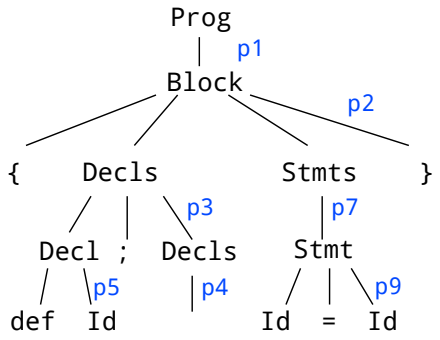
Top-Down vs. Bottom-Up Parsing

```
p1:  Prog  ::= Block
p2:  Block ::= '{' Decls Stmts '}'
p3:  Decls ::= Decl ';' Decls
p4:  Decls ::=
p5:  Decl  ::= 'def' Id
p6:  Stmts ::= Stmts ';' Stmt
p7:  Stmts ::= Stmt
p8:  Stmt  ::= Block
p9:  Stmt  ::= Id '=' Id
```

Grammar

```
Prog  ==> Block                          p1
      ==> { Decls Stmts }                p2
      ==> { Decl ; Decls Stmts }         p3
      ==> { def Id ; Decls Stmts }       p5
      ==> { def Id ; Stmts }             p4
      ==> { def Id ; Stmt }              p7
      ==> { def Id ; Id = Id }           p9
```

Top-Down, Leftmost Derivation



Derivation Tree

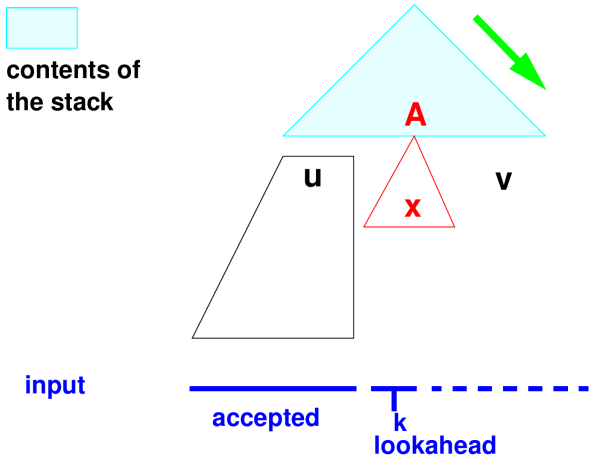
```
{ def Id ; Id = Id }       <==    p5
{ Decl ; Id = Id }         <==    p4
{ Decl ; Decls Id = Id }   <==    p3
{ Decls Id = Id }          <==    p9
{ Decls Stmt }             <==    p7
{ Decls Stmts }            <==    p2
Block                      <==    p1
Prog
```

Bottom-Up, Reverse Rightmost Derivation

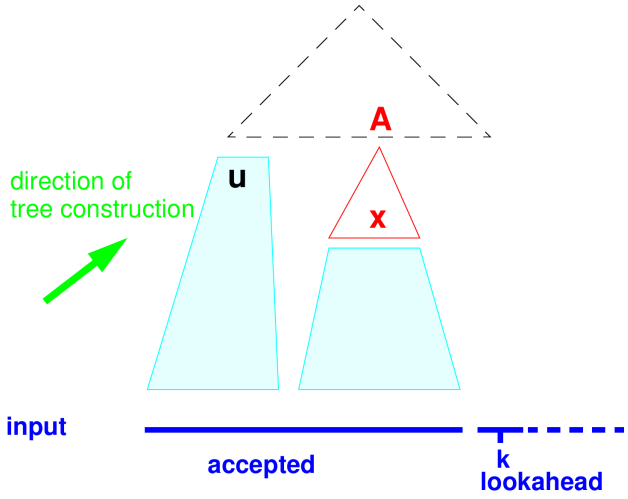
Top-Down vs. Bottom-Up Parsing (2)

Information a stack automaton has when it decides to apply production A ::= x :

top-down, predictive
leftmost derivation



bottom-up
rightmost derivation backwards



A bottom-up parser has seen more of the input when it decides to apply a production.

Consequence: **bottom-up** parsers and their grammar classes are more **powerful**.

Top-Down Parsing using Recursive Descent

A *Recursive Descent Parser* is constructed by systematically transforming the context-free grammar into a set of recursive functions:

Nonterminal `X`	⇒	Function `X` accepting `X`
Alternative productions for `X`	⇒	Branches in the function body
Decision set of production `p`	⇒	Set of terminal sequences indicating branch `p`
Nonterminal `Y` on the rhs	⇒	Call to function `Y`
Terminal occurrence `t`	⇒	*Accept* token `t`

The *Decision Set* of a production `p` is formally defined after the next slide.

The function `accept(t)` checks whether the current token is a `t`

- if yes, the next token is read
- if no, an error is reported

Recursive Descent Example

Productions for Stmt :

```
p1: Stmt ::= Variable '=' Expr
p2: Stmt ::= 'while' Expr Stmt
```

Function Stmt

```
void Stmt () {
   switch (CurrSymbol) {
      case decision set for p1:
         Variable();
         accept(assignSym);
         Expr();
         break;
      case decision set for p2:
         accept(whileSym);
         Expr();
         Stmt();
         break;
      default: Error();
   }
}
```


Grammar Conditions for Recursive Descent

A context-free grammar is **LL(1)** , if for any pair of productions that have the same symbol on their left-hand sides, `A::=u` and `A::=v` , the decision sets are disjoint.

$$\text{DecisionSet}(\texttt{A::=u}) \cap \text{DecisionSet}(\texttt{A::=v}) = \emptyset$$

with

- DecisionSet(`A::=u`) := if nullable (`u`) then First (`u`) ∪ Follow (`A`) else First (`u`)
- nullable (`u`) holds iff a derivation `u` $\Rightarrow^* \epsilon$ exists
- First (`u`) := {t ∈ T | v ∈ V^* exists and a derivation `u` $\Rightarrow^*$ t v }
- Follow (`A`) := {t ∈ T | u , v ∈ V^* exist, A ∈ N and a derivation S $\Rightarrow^*$ u A t v}

Productions	DecisionSet
p1: Prog ::= Block	{
p2: Block ::= '{' Decls Stmts '}'	{
p3: Decls ::= Decl ';' Decls	def
p4: Decls ::=	Id {
p5: Decl ::= 'def' Id	def
p6: Stmts ::= Stmts ';' Stmt	Id {
p7: Stmts ::= Stmt	Id {
p8: Stmt ::= Block	{
p9: Stmt ::= Id '=' Id	Id

not LL(1)

Computation Rules for nullable, First, and Follow

nullable(ε) = true; nullable(t) = false; nullable(uv) = nullable(u) ∧ nullable(v);
nullable(A) = true iff ∃ A::=u ∈ P ∧ nullable(u)

First(ε) = ∅; First(t) = {t};
First(uv) = if nullable(u) then First(u) ∪ First(v) else First(u)
First(A) = First(u_1) ∪...∪ First(u_n) for all A::=u_i ∈ P

Follow(A):
  if A=S then # ∈ Follow(A)
  if Y::=uAv ∈ P then First(v) ⊆ Follow(A) and if nullable(v) then Follow(Y) ⊆ Follow(A)

```
p1:  Prog  ::= Block
p2:  Block ::= '{' Decls Stmts '}'
p3:  Decls ::= Decl ';' Decls
p4:  Decls ::=
p5:  Decl  ::= 'def' Id
p6:  Stmts ::= Stmts ';' Stmt
p7:  Stmts ::= Stmt
p8:  Stmt  ::= Block
p9:  Stmt  ::= Id '=' Id
```

Nonterm	First	Follow
Prog	{	#
Block	{	# ; }
Decls	def	Id {
Decl	def	;
Stmts	Id {	; }
Stmt	Id {	; }

Grammar Transformations for LL(1)

Consequences of LL(1) condition

- Alternative productions must not begin with the same symbols.
- Productions must not be left-recursive, neither directly nor indirectly.

Grammar transformations that do not change the language

Left factorization:

Non-LL(1)

```
A ::= v u
A ::= v w
```

Transformed

```
A ::= v X
X ::= u
X ::= w
```

Elimination of direct left-recursion:

Non-LL(1)

```
A ::= A u
A ::= v
```

Transformed

```
A ::= v X
X ::= u X
X ::=
```

LL(1) Extension for EBNF Constructs

EBNF construct:	**Option [u]**	**Repetition (u)***
Production:	A ::= v [u] w	A ::= v (u)* w

additional LL(1)-condition:

if nullable(w)
then First(u) $\cap$ (First(w) $\cup$ Follow(A)) = $\emptyset$
else First(u) $\cap$ First(w) = $\emptyset$

in recursive descent parser:

```
v
if (CurrToken in First(u)) { u }
w
```

```
v
while (CurrToken in First(u)) { u }
w
```

Repetition (u)+ left as exercise