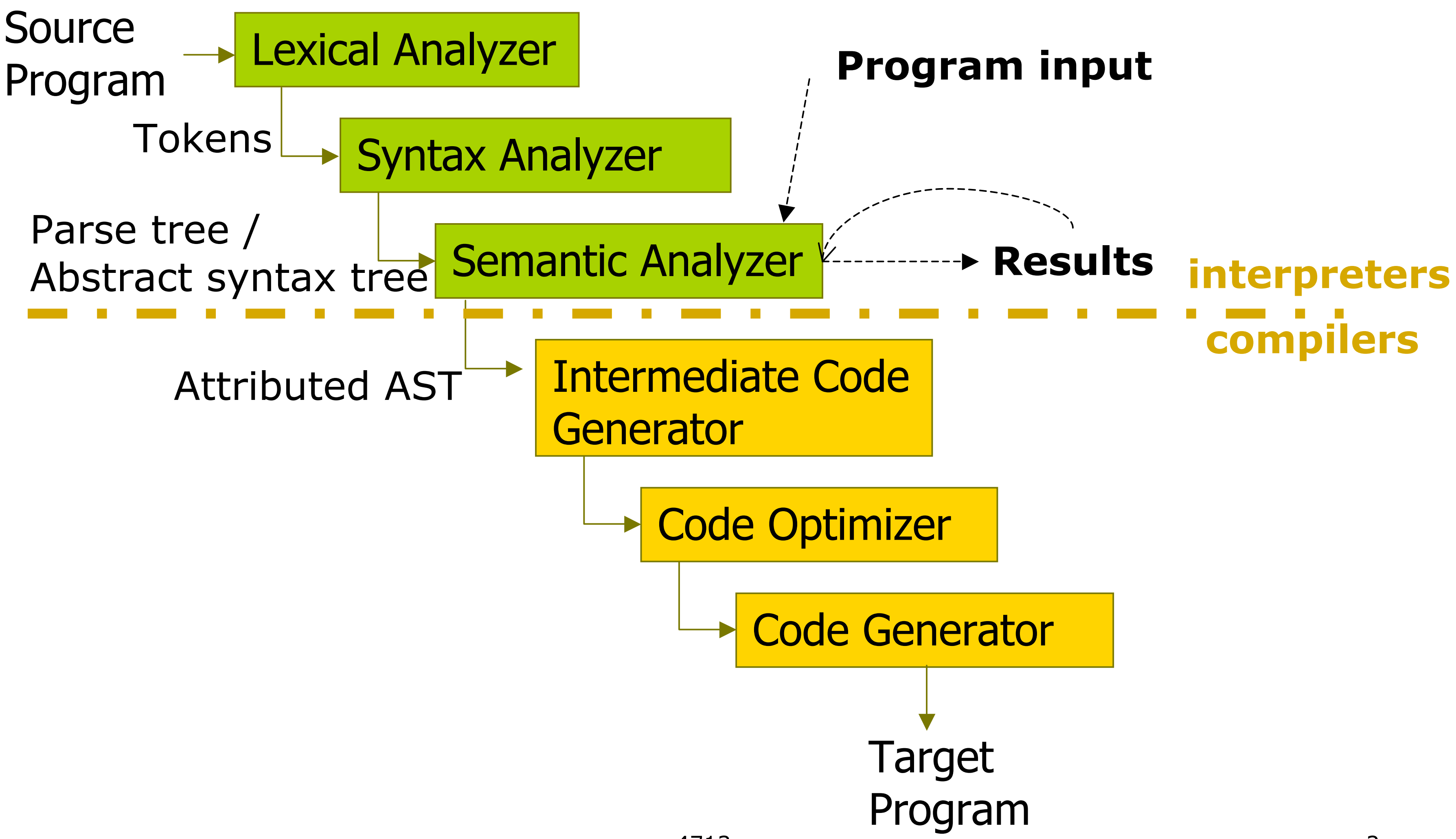
Syntax Directed Translation

Attribute grammar and translation schemes

Typical implementation of languages



Syntax-directed translation

- Compilers translate language constructs
  - Need to keep track of relevant information
    - Attributes: relevant information associated with a construct

e ::= n | e+e | e–e | e * e | e / e

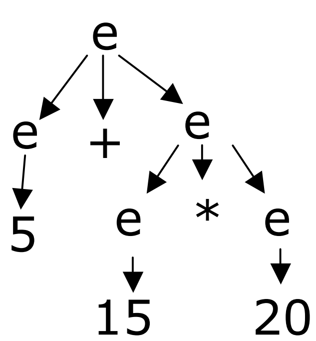
Attributes for expressions:
type of value: int, float, double, char, string,...
type of construct: variable, constant, operations, ...
Attributes for constants: values
Attributes for variables: name, scope
Attributes for operations: arity, operands, operator,...

Syntax directed definition

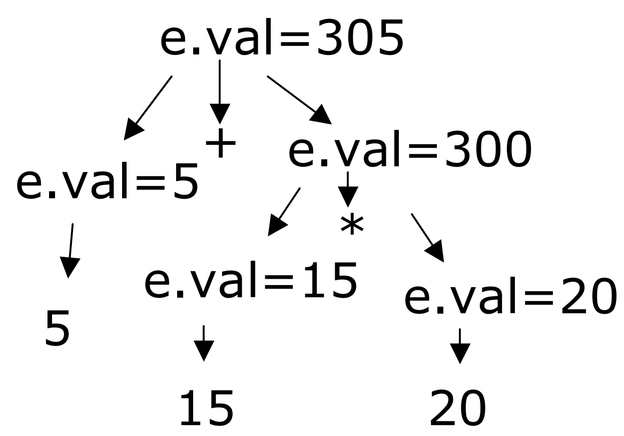
- Associate a set of attributes with each grammar symbol
- Associate a set of semantic rules with each production
  - Specify how to compute attribute values of symbols

e ::= n | e+e | e–e | e * e | e / e

Parse tree for 5 + 15 * 20:



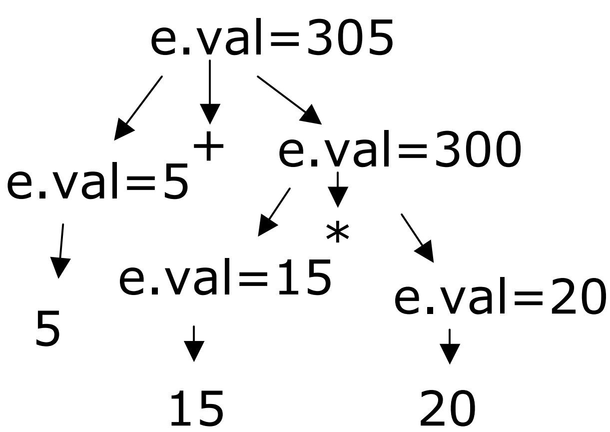
Annotated parse tree:



Synthesized attribute definition

- An attribute is synthesized if
  - The attribute value of parent is determined from attribute values of children in the parse tree

e ::= n | e+e | e–e | e * e | e / e

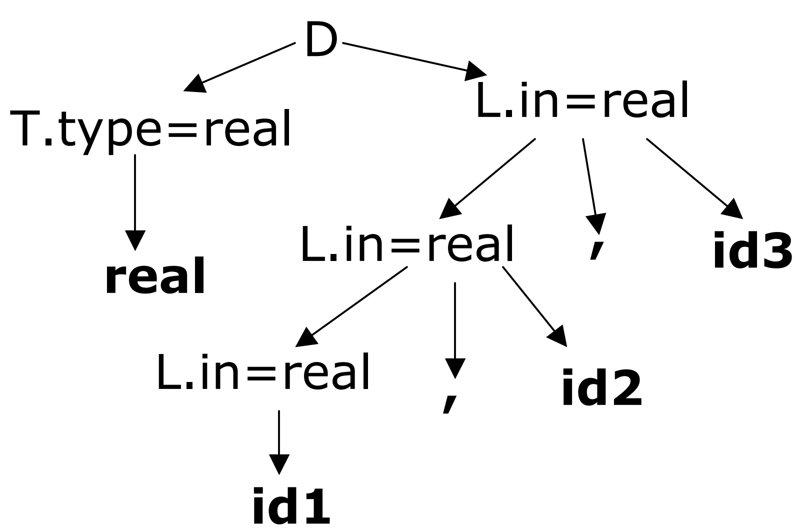


production	Semantic rules
e ::= n	e.val = n.val
e ::= e1 + e2	e.val = e1.val [+] e2.val
e ::= e1 - e2	e.val = e1.val [-] e2.val
e ::= e1 * e2	e.val = e1.val [*] e2.val
e ::= e1 / e2	e.val = e1.val [/] e2.val

Inherited attribute definition

- An attribute is inherited if
  - The attribute value of a parse-tree node is determined from attribute values of its parent and siblings

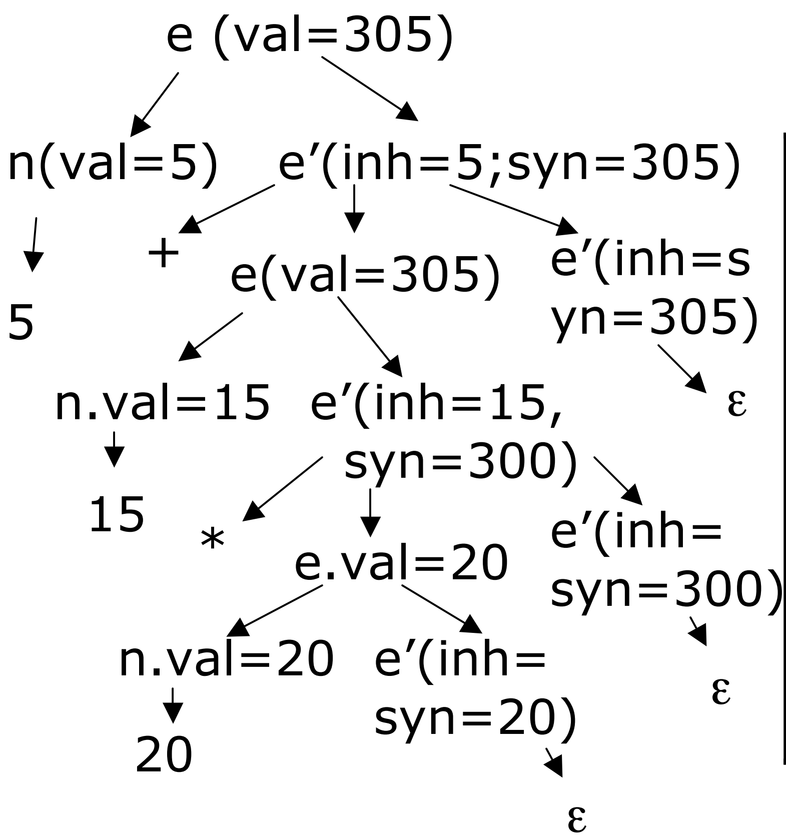
D::=T L
T::= **int** | **real**
L ::= L **,** **id** | **id**



Production	Semantic rules
D::=T L	L.in:=T.type
T::= **int**	T.Type:=integer
T::=**real**	T.type:=real
L::=L1 **,id**	L1.in := L.in Addtype(id.entry,L.in)
L::=**id**	Addtype(id.entry,L.in)

Synthesized and inherited attributes

- Sometimes both synthesized and inherited attributes are required to evaluate necessary information



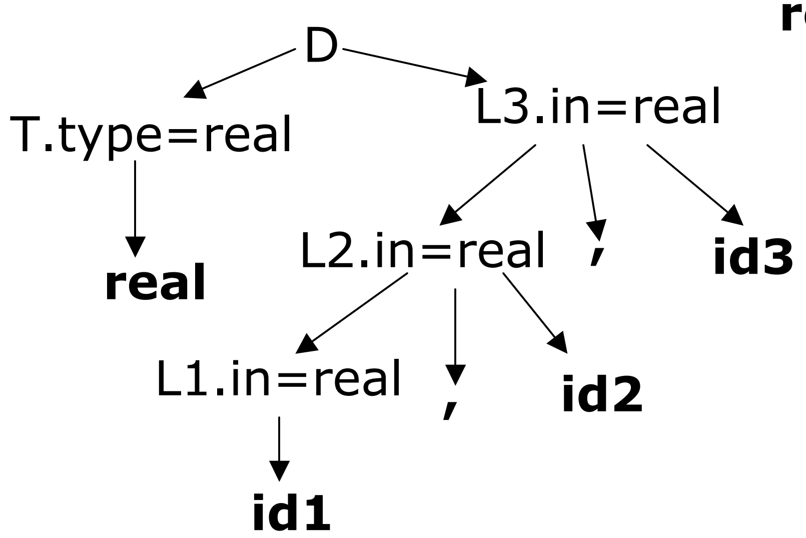
e ::= n e'
e' ::= +ee' | *ee' | ε

production	Semantic rules
e ::= n e'	e'.inh=n.val; e.val = e'.syn
e' ::= + e e'1	e'1.inh = e'.inh [+] e.val e'.syn = e'1.syn
e' ::= * e e'1	e'1.inh = e'.inh [*] e.val e'.syn = e'1.syn
e' ::= ε	e'.syn = e'.inh

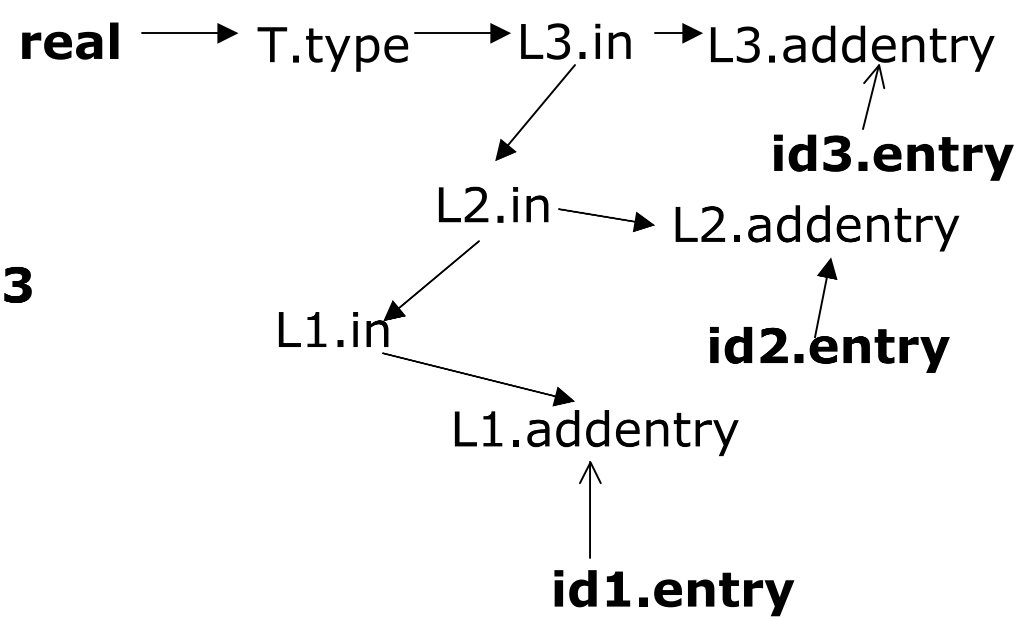
Dependences in semantic evaluation

- If value of attribute b depends on attribute c,
  - Semantic rule for b must be evaluated after semantic rule for c
  - There is a dependence from c to b

Annotated parse tree:

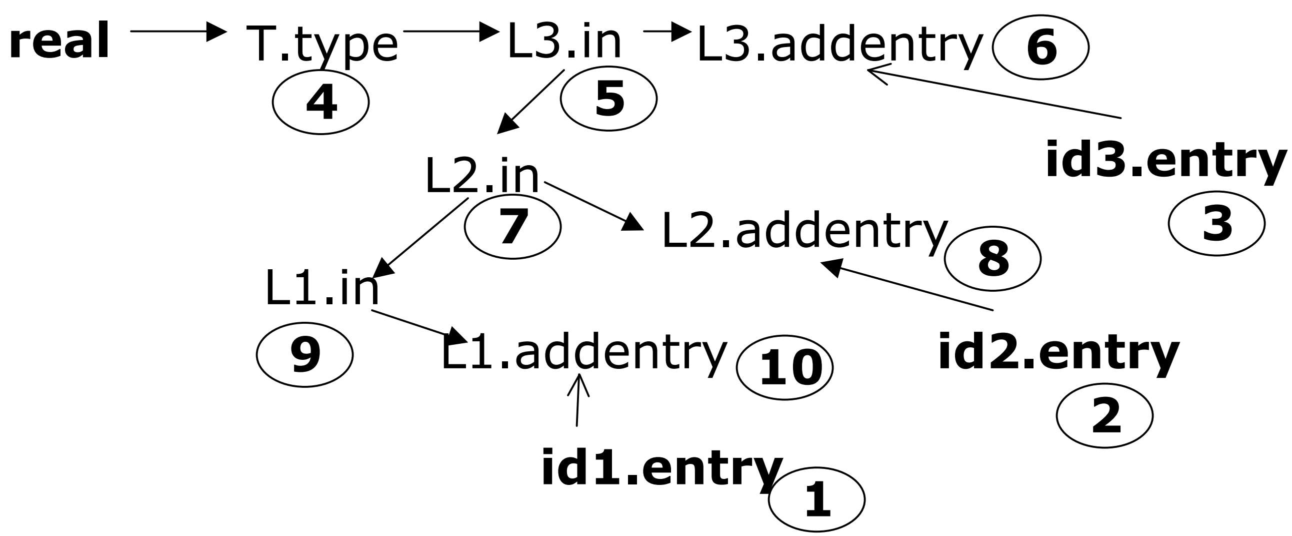


Dependency graph:



Evaluation order of semantics

- Topological order of the dependence graph
  - Edges go from nodes earlier in the ordering to later nodes
  - No cycles are allowed in dependence graph



Evaluation of semantic rules

- Parse-tree methods (compile time)
  - Build a parse tree for each input
  - Build a dependency graph from the parse tree
  - Obtain evaluation order from a topological order of the dependency graph
- Rule-based methods (compiler-construction time)
  - Predetermine the order of attribute evaluation for each production
- Oblivious methods
  - Evaluation order is independent of semantic rules
  - Evaluation order forced by parsing methods
  - Restrictive in acceptable attribute definitions

Bottom-up evaluation of attributes

- S-attributed definitions
  - Syntax-directed definitions with only synthesized attributes
  - Can be evaluated through post-order traversal of parse tree
- Synthesized attributes and bottom-up parsing
  - Keep attribute values of grammar symbols in stack
  - Evaluate attribute values at each reduction
- In top-down parsing, the return value of each parsing routine

Configuration of LR parser:

$$(\underbrace{s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m}_{\text{states}},\ \underbrace{a_i a_{i+1} \ldots a_n \$}_{\text{inputs}},\ \underbrace{v_1 v_2 \ldots v_m}_{\text{values}})$$

Right-sentential form: $X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n \$$

Automata states: $s_0 s_1 s_2 \ldots s_m$

Grammar symbols in stack: X1X2…Xm

Synthesized attribute values of $X_i \rightarrow v_i$

Implementing S-attributed definitions

Implementation of a desk calculator with an LR parser (when a number is shifted onto symbol stack, its value is shifted onto val stack)

production	Code fragment
E’ ::= E	Print(val[top])
E ::= E1 + T	v=val[top-2]+val[top]; top-=2; val[top]=v;
E ::= T	
T ::= T1 * F	v=val[top-2]*val[top]; top-=2; val[top]=v;
T ::= F	
F ::= (E)	v=val[top-1]; top-=2; val[top]=v
F ::= n	

L-attributed definitions

- A syntax-directed definition is L-attributed if each inherited attribute of X_j, $1<=j<=n$, on the right side of $A::=X_1X_2\ldots X_n$, depends only on
  - the attributes of $X_1,X_2,\ldots,X_{j-1}$ to the left of X_j in the production
  - the inherited attributes of A

L-attributed definition

Production	Semantic rules
D::=T L	L.in:=T.type
T::= **int**	T.Type:=integer
T::=**real**	T.type:=real
L::=L1 **,id**	L1.in := L.in Addtype(id.entry,L.in)
L::=**id**	Addtype(id.entry,L.in)

Non L-attributed definition

Production	Semantic rules
A::=L M	L.i = A.i M.i = L.s A.s = M.s
A ::= Q R	R.i = A.i Q.i = R.s A.s = Q.s

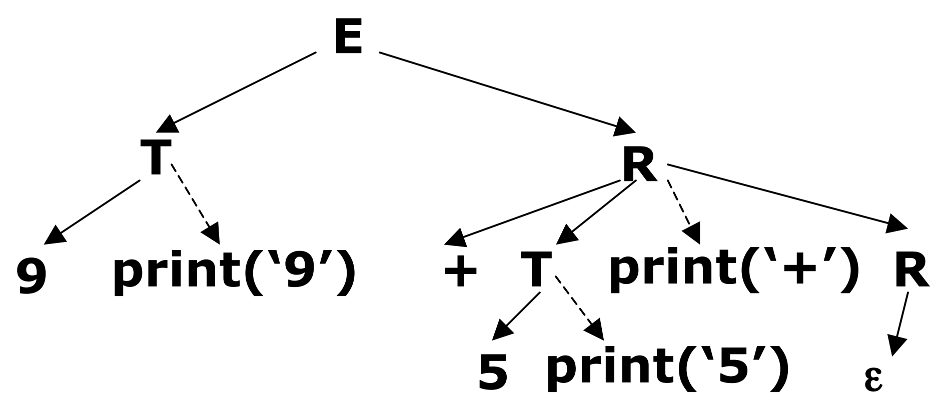
Translation schemes

- A translation scheme is a CFG where
  - Attributes are associated with grammar symbols and
  - Semantic actions are inserted within right sides of productions
- Notation for specifying translation during parsing

Translation scheme:

```
E ::= T R
R ::= '+' T {print('+')} R1
      | ε
T ::= num {print(num.val)}
```

Parse tree for 9+5 with actions



Treat actions as though they are terminal symbols.

Designing translation schemes

- How to compute attribute values at each production?

D::=T L	L.in:=T.type
T::= **int**	T.Type:=integer
T::=**real**	T.type:=real
L::= **id, L1**	L1.in := L.in; Addtype(id.entry,L.in)
L::=**id**	Addtype(id.entry,L.in)

- Every attribute value must be available when referenced
  - S-attribute of left-hand symbol computed at end of production
  - I-attribute of right-hand symbol computed before the symbol
  - S-attribute of right-hand symbol referenced after the symbol

```
D::=T { L.in:=T.type} L
T::= int  {T.Type:=integer}
T::=real  { T.type:=real}
L::= id , {Addtype(id.entry,L.in) } {L1.in := L.in} L1
L::=id  {Addtype(id.entry,L.in)}
```

Top-down translation

```
void parseD()
   { Type t = parseT(); }
      parseL(t);
    }
Type parseT
     {  switch (currentToken()) {
         case INT: return TYPE_INT;
         case REAL:  return TYPE_REAL;
         }
     }
void parseL(Type in)
     {
       SymEntry e = parseID();
       AddType(e, in);
       if (currentToken() == COMMA) {
          parseTerminal(COMMA);
          parseL(in)
      }
}
```

Top-down translation

- For each non-terminal A, construct a function that
  - Has a formal parameter for each inherited attribute of A
  - Returns the values of the synthesized attributes of A
- The code associated with each production does the following
  - Save the s-attribute of each token X into a variable X.x
  - Generate an assignment B.s=parseB(B.i1,B.i2,…,B.ik) for each non-terminal B, where B.i1,…,B.ik are values for the L-attributes of B and B.s is a variable to store s-attributes of B.
  - Copy the code for each action, replacing references to attributes by the corresponding variables

Bottom-up translation in Yacc

```
D::=T { L.in:=T.type} L
T::= int  {T.Type:=integer}
T::=real  { T.type:=real}
L::= {L1.in := L.in} L1,id {Addtype(id.entry,L.in) }
L::=id  {Addtype(id.entry,L.in)}
```

↓

```
D : T  {$$ = $1; } L
T : INT { $$ = integer; } | REAL { $$ = real; }
L : L COMMA ID { Addtype($3, $0); }
    | ID { Addtype($1,$0); }
```