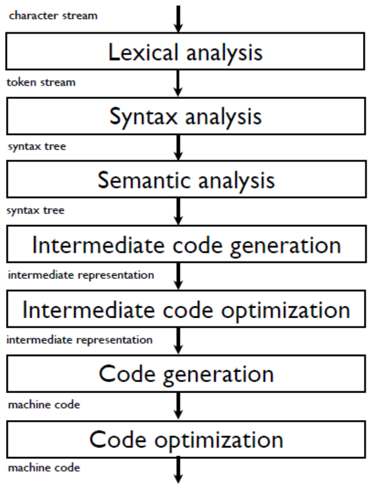
# Course Name: M.Sc. in Computer Science
# Semester – IV, Session: 2018-2020
# Department of Computer Science
# Name of Faculty: Gautam Mahapatra, Associate Professor
# Subject: Compiler Design
# Class Taken:
# Date: 24th April 2020, Time: 4.15PM – 6.00PM

# Number of Students Attended: 8

# Software used: Skype

# Details of the subject taught: Syntax Directed Translation

## Structure of a compiler




## Definitions

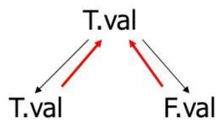
- Syntax-directed definition (Attribute Grammar)
  - Productions with semantic rules
    - Ex: E→E1+T    E.code=E1.code | T.code | '+'
  - More readable
  - Useful for specification
- Syntax-directed translation (Translation Scheme)
  - Productions with semantic actions
    - Ex: E→E1+T    { print '+' }
  - More efficient
  - Useful for implementation

## Syntax-Directed Definitions

- SDD: a context-free grammar with attributes and rules
  - Attributes: for grammar symbols
  - Rules: for productions
- Attributes for nonterminals
  - *Synthesized attribute*: attributes that are passed up a parse tree, i.e., the LHS attribute is computed from the RHS attributes in the production rules
    - $E \rightarrow E_1 + T$    $E.val = E_1.val + T.val$
  - *Inherited attribute*: attributes that are passed down a parse tree, i.e., the RHS attributes are derived from its LHS attributes or other RHS attributes in the production rules
    - $T \rightarrow F\ T'$    $T'.inh = F.val$  $T.val = T'.syn$
- Terminals can have synthesized attributes, but not inherited attributes

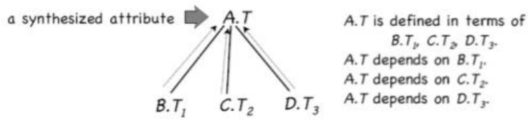
# Synthesized Attributes

- The attribute value of the terminal at the left hand side of a grammar rule depends on the values of the attributes on the right hand side.
- Typical for LR (bottom up) parsing.
- Example: T→T*F {$$.val=$1.val×$3.val}.



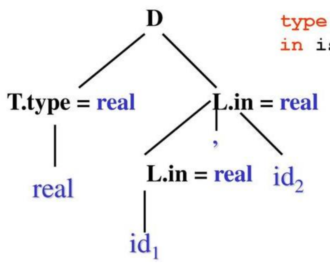

## Synthesized attributes

- Synthesized attributes depend only on the attributes of children. They are the most common attribute type.




## Example of Inherited Attributes

| | |
|---|---|
| D → T L | { L.in := T.type } |
| T → int | {T.type := integer } |
| T → real | {T.type := real } |
| L → $L_1$ , id | {$L_1$.in := L.in ; addtype (id.entry, L.in)} |
| L → id | {addtype (id.entry, L.in)} |




type is a synthesized attribute
in is an inherited attribute

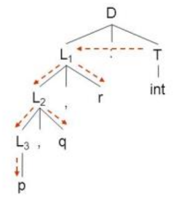
## Inherited Attribute

- Example: not an L-attributed grammar

| | |
|---|---|
| D → L : T | L.type = T.type |
| L → $L_1$, id | $L_1$.type = L.type, addtype (id.entry, L.type) |
| L → id | addtype (id.entry, L.type) |
| T → int | T.type = integer |
| T → real | T.type = real |

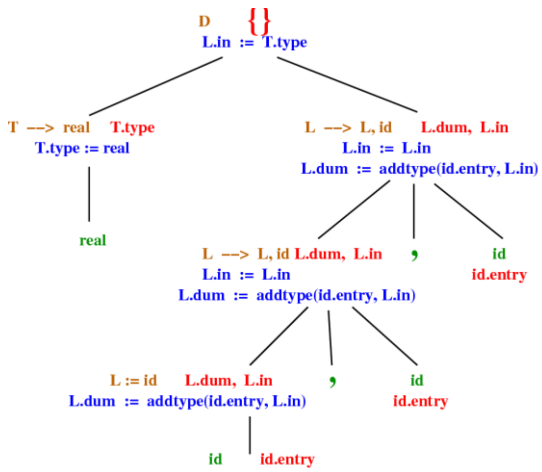
Depend on right sibling
No longer L-attributed

- Parse p, q, r : int
- Inherited attribute
- Needs two-pass evaluation
  - First build the parse tree
  - Then evaluate




## Syntax Directed Translation

- Translation is directed by the syntax, or the structure of the program.
- Can be done
  - Directly, while parsing.
  - From the structure of an abstract syntax tree.
- Syntax directed translation attaches a meaning to every production (or piece of syntax)
  - This meaning is often given in terms of the meaning of the sub-expressions
- Often we think of this meaning being an attribute of each syntax node.
- Attribute grammars provide a natural way of describing this.



# Syntax-Directed Translation

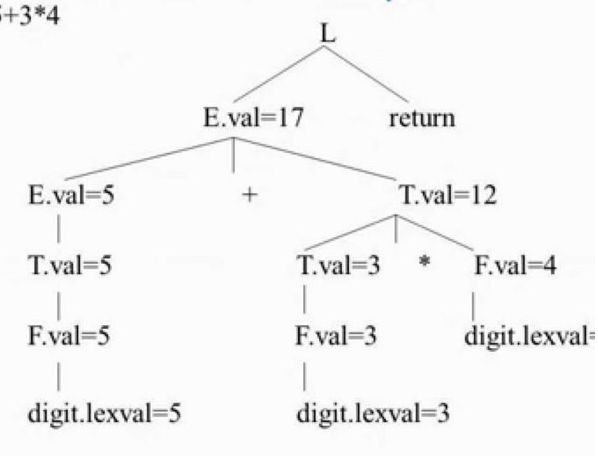
- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- Evaluation of these semantic rules:
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities
  - in fact, they may perform almost any activities.
- An attribute may hold almost any thing.
  - a string, a number, a memory location, a complex record.

## Syntax-Directed Translation of Abstract Syntax Trees

| Production | Semantic Rule |
|---|---|
| $S \rightarrow$ **id** **:=** $E$ | $S$.nptr := *mknode*(':=', *mkleaf*(**id**, **id**.entry), $E$.nptr) |
| $E \rightarrow E_1$ **+** $E_2$ | $E$.nptr := *mknode*('+', $E_1$.nptr, $E_2$.nptr) |
| $E \rightarrow E_1$ ***** $E_2$ | $E$.nptr := *mknode*('*', $E_1$.nptr, $E_2$.nptr) |
| $E \rightarrow$ **-** $E_1$ | $E$.nptr := *mknode*('uminus', $E_1$.nptr) |
| $E \rightarrow$ **(** $E_1$ **)** | $E$.nptr := $E_1$.nptr |
| $E \rightarrow$ **id** | $E$.nptr := *mkleaf*(**id**, **id**.entry) |

## Annotated Parse Tree -- Example

Input: 5+3*4



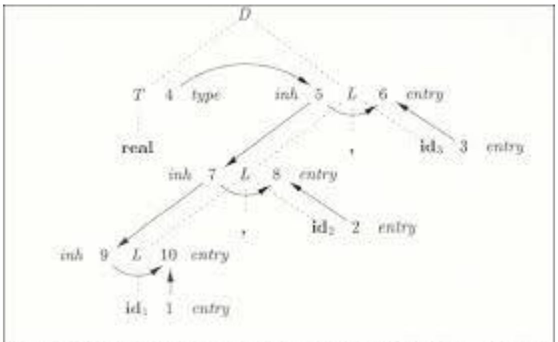



Figure 5.9: Dependency graph for a declaration **float** $id_1$ , $id_2$ , $id_3$

| | PRODUCTION | SEMANTIC RULES |
|---|---|---|
| 1) | $L \to E\ \mathbf{n}$ | $L.val = E.val$ |
| 2) | $E \to E_1 + T$ | $E.val = E_1.val + T.val$ |
| 3) | $E \to T$ | $E.val = T.val$ |
| 4) | $T \to T_1 * F$ | $T.val = T_1.val \times F.val$ |
| 5) | $T \to F$ | $T.val = F.val$ |
| 6) | $F \to ( E )$ | $F.val = E.val$ |
| 7) | $F \to \mathbf{digit}$ | $F.val = \mathbf{digit}.\text{lexval}$ |

| | |
|---|---|
| $L \to E\ \mathbf{n}$ | { print ( $E.val$ ); } |
| $E \to E_1 + T$ | { $E.val = E_1.val + T.val$; } |
| $E \to T$ | { $E.val = T.val$; } |
| $T \to T_1 * F$ | { $T.val = T_1.val \times F.val$; } |
| $T \to F$ | { $T.val = F.val$; } |
| $F \to ( E )$ | { $F.val = E.val$; } |
| $F \to \mathbf{digit}$ | { $F.val = \mathbf{digit}.lexval$; } |

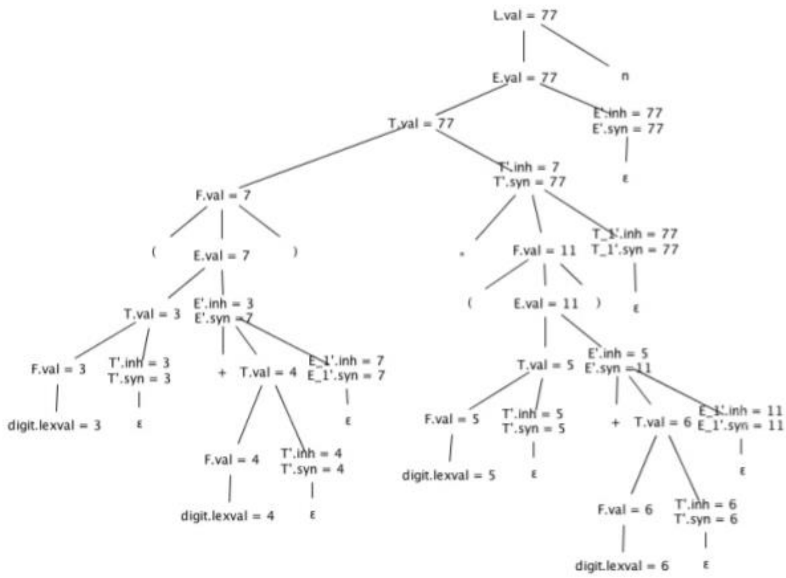
| PRODUCTION | SEMANTIC RULE |
|---|---|
| $expr \to expr_1 + term$ | $expr.t := expr_1.t \Vert term.t \Vert '+'$ |
| $expr \to expr_1 - term$ | $expr.t := expr_1.t \Vert term.t \Vert '-'$ |
| $expr \to term$ | $expr.t := term.t$ |
| $term \to 0$ | $term.t := '0'$ |
| $term \to 1$ | $term.t := '1'$ |
| ... | ... |
| $term \to 9$ | $term.t := '9'$ |

| PRODUCTION | SEMANTIC RULE |
|---|---|
| $L \to E\text{n}$ | $print(E.val)$ |
| $E \to E_1 + T$ | $E.val := E_1.val + T.val$ |
| $E \to T$ | $E.val := T.val$ |
| $T \to T_1 * F$ | $T.val := T_1.val * F.val$ |
| $T \to F$ | $T.val := F.val$ |
| $F \to (E)$ | $F.val := E.val$ |
| $F \to \text{digit}$ | $F.val :=\text{digit}.lexval$ |

| Production | Semantic Rules |
|---|---|
| S → **id** := E | S.code := E.code $\Vert$ gen(**id**.place ':=' E.place) |
| E → $E_1$ + $E_2$ | E.place := newtemp;<br>E.code := $E_1$.code $\Vert$ $E_2$.code $\Vert$<br>gen(E.place ':=' $E_1$.place '+' $E_2$.place) |
| E → $E_1$ * $E_2$ | E.place := newtemp;<br>E.code := $E_1$.code $\Vert$ $E_2$.code $\Vert$<br>gen(E.place ':=' $E_1$.place '*' $E_2$.place) |

| Production | Semantic Rules |
|---|---|
| 1) $E \to T\ E'$ | $E.node = E'.syn$<br>$E'.inh = T.node$ |
| 2) $E' \to +T\ E_1'$ | $E_1'.inh$ = **new** *Node* ( '+', $E'.inh$, $T.node$ )<br>$E'.syn = E_1'.syn$ |
| 3) $E' \to -T\ E_1'$ | $E_1'.inh$ = **new** *Node* ( '−', $E'.inh$, $T.node$ )<br>$E'.syn = E_1'.syn$ |
| 4) $E' \to \varepsilon$ | $E'.syn = E'.inh$ |
| 5) $T \to ( E )$ | $T.node = E.node$ |
| 6) $T \to$ **id** | $T.node$ = **new** *Leaf* ( **id**, **id**.*entry* ) |
| 7) $T \to$ **num** | $T.node$ = **new** *Leaf* ( **num**, **num**.*val* ) |

| | PRODUCTION | SEMANTIC RULES |
|---|---|---|
| 1) | $D \to T\ L$ | $L.inh = T.type$ |
| 2) | $T \to \mathbf{int}$ | $T.type = \text{integer}$ |
| 3) | $T \to \mathbf{float}$ | $T.type = \text{float}$ |
| 4) | $L \to L_1\ ,\ \mathbf{id}$ | $L_1.inh = L.inh$<br>$addType(\mathbf{id}.entry, L.inh)$ |
| 5) | $L \to \mathbf{id}$ | $addType(\mathbf{id}.entry, L.inh)$ |




Along with the text book : Principles of Compiler Design, by Aho, Sehti and Ullman.