Syntax-Directed Translation

Technique used to build semantic information for large structures, based on its syntax.

In a compiler, *Syntax-Directed Translation* is used for

- Constructing Abstract Syntax Tree
- Type checking
- Intermediate code generation

The Essence of Syntax-Directed Translation

The semantics (*i.e.*, *meaning*) of the various constructs in the language is viewed as *attributes* of the corresponding grammar symbols.
Example:

sequence of characters 495
→ *grammar symbol* `TOK_INT`
→ *meaning* ≡ *integer 495*
→ *is an attribute of* `TOK_INT` (`yylval.int_val`).

Attributes are associated with **Terminal** as well as **Nonterminal** symbols.

An Example of Syntax-Directed Translation

$$\boxed{\begin{array}{lcl} E & \longrightarrow & E \texttt{ * } E \\ E & \longrightarrow & E \texttt{ + } E \\ E & \longrightarrow & \texttt{id} \end{array}}$$

$$\begin{array}{lcll} E & \longrightarrow & E_1 \texttt{ * } E_2 & \{E.val := E_1.val * E_2.val\} \\ E & \longrightarrow & E_1 \texttt{ + } E_2 & \{E.val := E_1.val + E_2.val\} \\ E & \longrightarrow & \texttt{int} & \{E.val := \texttt{int}.val\} \end{array}$$

Another Example of Syntax-Directed Translation

Decl	$\longrightarrow$	*Type VarList*
Type	$\longrightarrow$	`integer`
Type	$\longrightarrow$	`float`
VarList	$\longrightarrow$	`id ,` *VarList*
VarList	$\longrightarrow$	`id`

Decl	$\longrightarrow$	*Type VarList*	$\{VarList.type := Type.type\}$
Type	$\longrightarrow$	`integer`	$\{Type.type := int\}$
Type	$\longrightarrow$	`float`	$\{Type.type := float\}$
VarList	$\longrightarrow$	`id ,` $VarList_1$	$\{VarList_1.type := VarList.type;$ $\texttt{id}.type := VarList.type\}$
VarList	$\longrightarrow$	`id`	$\{\texttt{id}.type := VarList.type\}$

Attributes

- **Synthesized:** Attribute of LHS symbol of a grammar rule, whose value depends on attributes of RHS symbols of the grammar rule.
  - Value flows from child to parent in the parse tree.
  - Example: *val* in Expression grammar
- **Inherited:** Attribute of an RHS symbol of a grammar rule, whose value depends on attributes of the LHS symbol and the other RHS symbols of the grammar rule.
  - Value flows into a node in the parse tree from parents and/or siblings.
  - Example: *type* of *VarList* in declaration grammar

Syntax-Directed Definition

Actions associated with each production in a grammar.
For a production $A \longrightarrow X\ Y$, actions may be of the form:

- $A.attr := f(X.attr', Y.attr'')$ for synthesized attributes
- $Y.attr := f(A.attr', X.attr'')$ for inherited attributes

If the function f does not have side effects, syntax directed definitions is also called as *attribute grammars*.

Attributes and Definitions

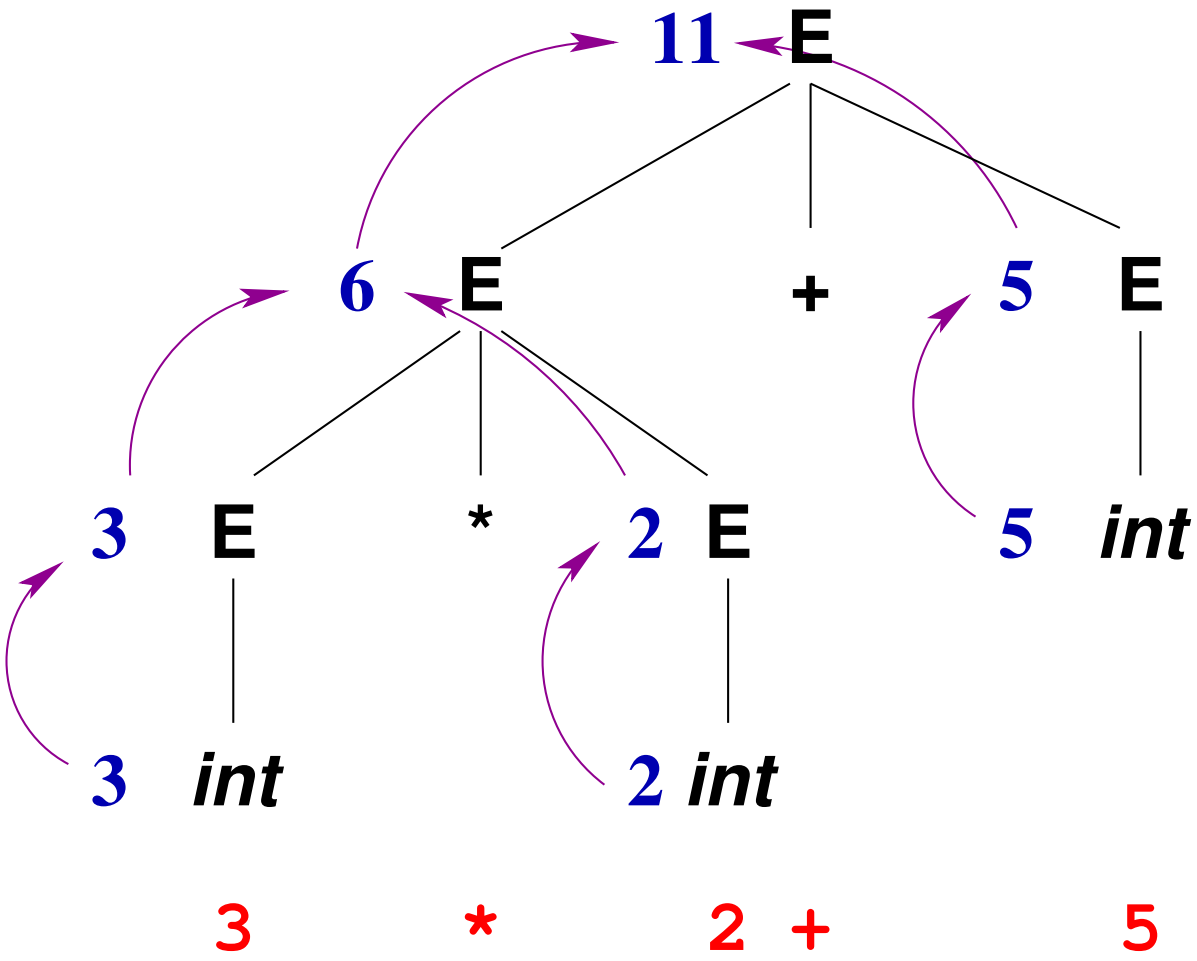
- **S-Attributed Definitions:** Where all attributes are *synthesized*.
- **L-Attributed Definitions:** Where all *inherited* attributes are such that their values depend only on
  - inherited attributes of the parent, and
  - attributes of left siblings

Synthesized Attributes: An Example

$$\boxed{\begin{array}{lcl} E & \longrightarrow & E \texttt{ * } E \\ E & \longrightarrow & E \texttt{ + } E \\ E & \longrightarrow & \texttt{int} \end{array}}$$

$$\begin{array}{lcll} E & \longrightarrow & E_1 \texttt{ * } E_2 & \{E.val := E_1.val * E_2.val\} \\ E & \longrightarrow & E_1 \texttt{ + } E_2 & \{E.val := E_1.val + E_2.val\} \\ E & \longrightarrow & \texttt{int} & \{E.val := \texttt{int}.val\} \end{array}$$

Information Flow for "Expression" Example

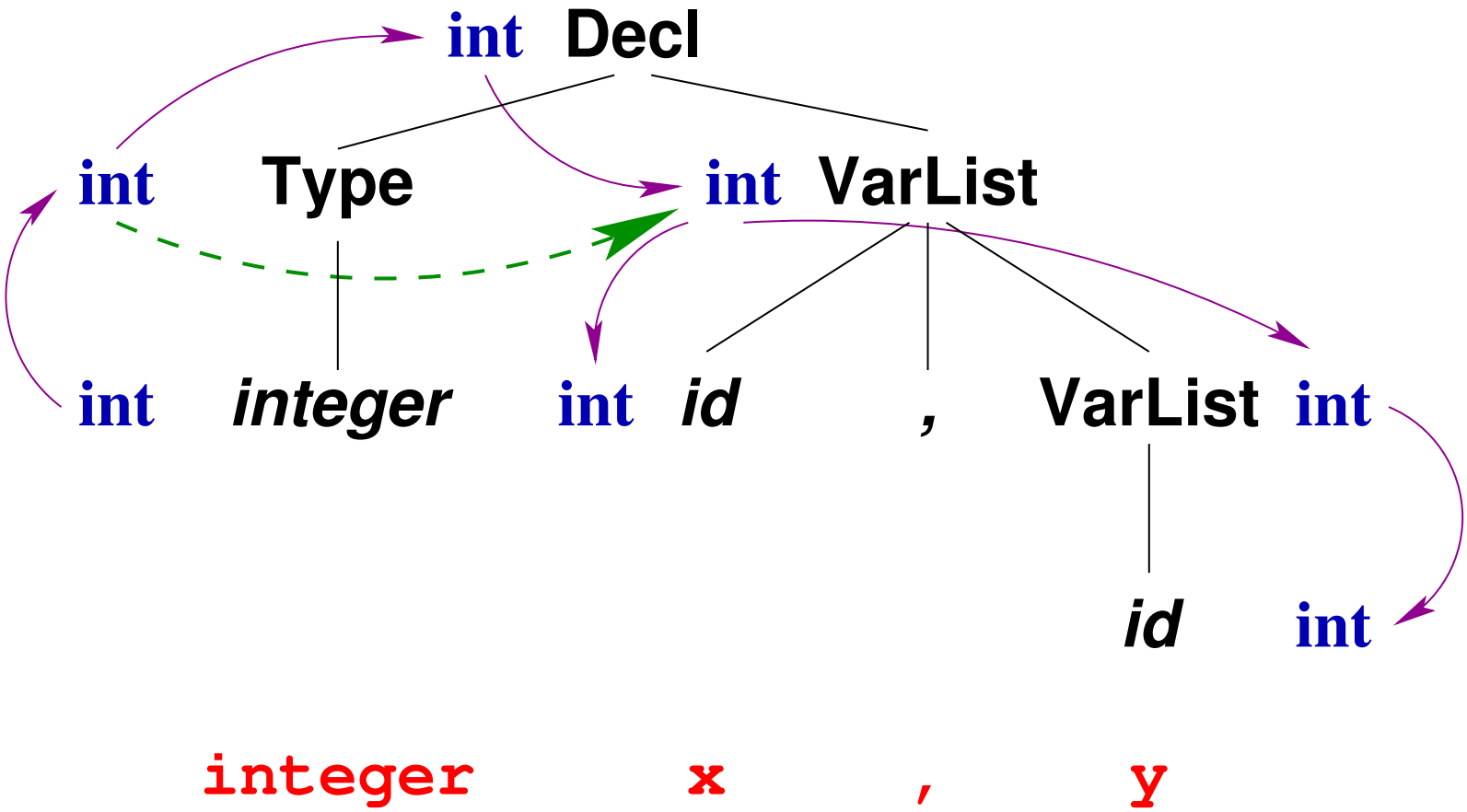


Another Example of Syntax-Directed Translation

Decl	$\longrightarrow$	*Type VarList*
Type	$\longrightarrow$	`integer`
Type	$\longrightarrow$	`float`
VarList	$\longrightarrow$	`id ,` *VarList*
VarList	$\longrightarrow$	`id`

Decl	$\longrightarrow$	*Type VarList*	$\{VarList.type := Type.type\}$
Type	$\longrightarrow$	`integer`	$\{Type.type := int\}$
Type	$\longrightarrow$	`float`	$\{Type.type := float\}$
VarList	$\longrightarrow$	`id ,` $VarList_1$	$\{VarList_1.type := VarList.type;$ $\texttt{id}.type := VarList.type\}$
VarList	$\longrightarrow$	`id`	$\{\texttt{id}.type := VarList.type\}$

Information Flow for "Type" Example



Syntax-Directed Definitions with `yacc`/`Bison`

E	$\longrightarrow$	E_1 `*` E_2	$\{E.val := E_1.val * E_2.val\}$
E	$\longrightarrow$	E_1 `+` E_2	$\{E.val := E_1.val + E_2.val\}$
E	$\longrightarrow$	`int`	$\{E.val :=$ `int`$.val\}$

E	:	E `MULT` E	$\{\$\$.val = \$1.val * \$3.val\}$
E	:	E `PLUS` E	$\{\$\$.val = \$1.val + \$3.val\}$
E	:	`INT`	$\{\$\$.val = \$1.val\}$

Syntax-Directed Definitions with PLY

E	$\longrightarrow$	E_1 * E_2	$\{E.val := E_1.val * E_2.val\}$
E	$\longrightarrow$	E_1 + E_2	$\{E.val := E_1.val + E_2.val\}$
E	$\longrightarrow$	int	$\{E.val :=$ int$.val\}$

```
def p_e_star(p):
    '''e : e '*' e'''
    p[0] = p[1] * p[3]

def p_e_plus(p):
    '''e : e '+' e'''
    p[0] = p[1] + p[3]

def p_e_int(p):
    '''e : int'''
    p[0] = p[1]
```

Synthesized Attributes and Bottom-up Parsing

Keep track of attributes of symbols while parsing.

- **Shift-reduce parsing:** Keep a stack of attributes corresponding to stack of symbols.
  Compute attributes of LHS symbol while performing reduction (*i.e.*, while pushing the symbol on symbol stack)

Synthesized Attributes & Shift-reduce parsing

E	$\longrightarrow$	E+E
E	$\longrightarrow$	E*E
E	$\longrightarrow$	int

Stack	Input Stream	Attributes
$	3 * 2 + 5 $	$
$ int	* 2 + 5 $	$ 3
$ E	* 2 + 5 $	$ 3
$ E *	2 + 5 $	$ 3 ⊥
$ E * int	+ 5 $	$ 3 ⊥ 2
$ E	+ 5 $	$ 6
$ E +	5 $	$ 6 ⊥
$ E + int	$	$ 6 ⊥ 5
$ E + E	$	$ 6 ⊥ 5
$ E	$	$ 11

Inherited Attributes

Ss	$\longrightarrow$	S ; Ss
Ss	$\longrightarrow$	ϵ
B	$\longrightarrow$	{ Ss }
S	$\longrightarrow$	B
S	$\longrightarrow$	other

Ss	$\longrightarrow$	S ; Ss_1	$\{S.block = Ss.block;$ $Ss_1.block = Ss.block;\}$
Ss	$\longrightarrow$	ϵ	
B	$\longrightarrow$	{ Ss }	$\{Ss.block = child(B.block);\}$
S	$\longrightarrow$	B	$\{B.block = S.block;\}$
S	$\longrightarrow$	other	$\{\texttt{other}.block = S.block;\}$

Top-down Parsing

```
parse_Ss() {
  /* production 1 */
    parse_S();
    parse_Ss();
    return;

  /* production 2 */
    return;
}
parse_B() {
  consume(OPEN_BRACE);
  parse_Ss();
  consume(CLOSE_BRACE);
}
```

Inherited Attributes and Top-down Parsing

```
parse_Ss(block) {
  /* production 1 */
    parse_S(block);
    parse_Ss(block);
    return;

  /* production 2 */
    return;
}
parse_B(block) {
  consume(OPEN_BRACE);
  parse_Ss(child(block));
  consume(CLOSE_BRACE);
}
```

Attributes and Top-down Parsing

- Inherited: analogous to function arguments
- Synthesized: analogous to return values

L-attributed definitions mean that argument to a parsing function is

- argument of the calling function, or
- return value/argument of a previously called function

Attributes and Bottom-up Parsing

- Synthesized: stored along with symbols on stack
- Inherited: **???**

Inherited Attributes and Bottom-up Parsing

Inherited attributes depend on the *context* in which a symbol is used.
For inherited attributes, we cannot assign an value to a node's attributes unless the parent's attributes are known.
When building parse trees bottom-up, parent of a node is not known when the node is created!

Solution:

- *Ensure that all attributes are inherited only from left siblings.*
- *Use "global" variables to capture inherited values,*
- *and introduce "marker" nonterminals to manipulate the global variables.*

Inherited Attributes & Bottom-up parsing

Ss	$\longrightarrow$	S ; Ss
Ss	$\longrightarrow$	ϵ
B	$\longrightarrow$	{ Ss }
S	$\longrightarrow$	B
S	$\longrightarrow$	other

B	$\longrightarrow$	{	M_1	Ss	M_2	}	
M_1	$\longrightarrow$	ϵ					{ $current\_block{+}{+}$;}
M_2	$\longrightarrow$	ϵ					{ $current\_block{-}{-}$;}

M_1 and M_2 are marker non-terminals.