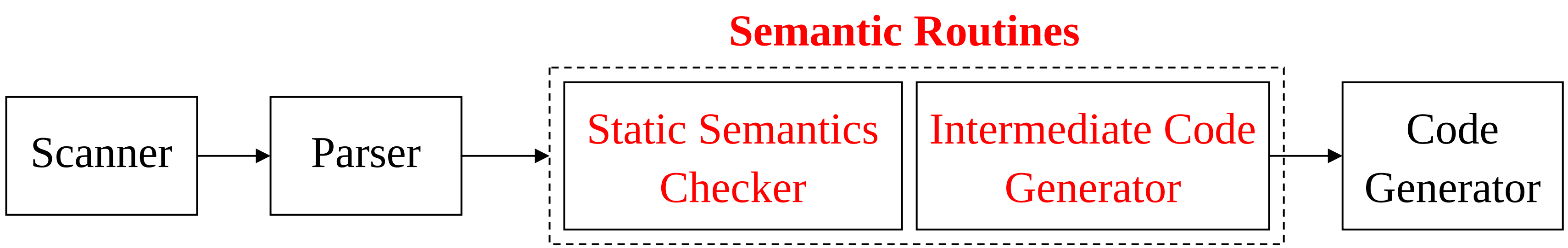
Semantic Processing

- ❖ The Semantic Processing phase consists of:
  - ✸ Checking the **Static Semantics** of the language
  - ✸ Generating an **Intermediate Representation** of the program
- ❖ Checking the static semantics include:
  - ✸ Making sure that all identifiers used in a program are declared
  - ✸ Making sure that all functions called are declared or defined
  - ✸ Making sure that parameters are passed correctly
  - ✸ Checking the uses of operators and types of expressions
  - ✸ Entering identifiers in symbol tables



Attribute Grammars

- Provides a practical formalism for describing semantic processing
- Proposed by Knuth in 1968
- Each grammar symbol has an associated set of **attributes**
- An **attribute** can represent anything we choose
  - The value of an expression when literal constants are used
  - The data type of a constant, variable, or expression
  - The location (or offset) of a variable in memory
  - The translated code of an expression, statement, or function
- An **annotated** or **attributed** parse tree is a
  - Parse tree showing the values of attributes at each node
- Attributes may be evaluated on the fly as an input is parsed
- Alternatively, attributes may be also evaluated after parsing

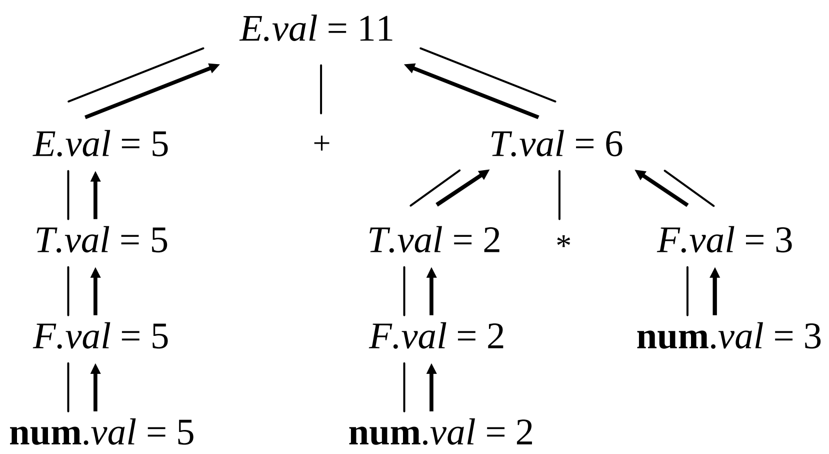
Synthesized and Inherited Attributes

- The attributes are divided into two classes:
  - **Synthesized** Attributes
  - **Inherited** Attributes
- A **synthesized attribute** of a parse tree node is computed from
  - Attribute values of the **children nodes**
- An **inherited attribute** of a parse tree node is computed from
  - Attribute values of the **parent node**
  - Attribute values of the **sibling nodes**
- Tokens may have only synthesized attributes
  - Token attributes are supplied by the scanner
- Nonterminals may have synthesized and/or inherited attributes
- Attributes are evaluated according to **Semantic rules**
  - Semantic rules are associated with production rules

S-Attributed Grammars

- S-Attributed grammars allow only synthesized attributes
- Synthesized attributes are evaluated bottom up
- S-Attributed grammars work perfectly with LR parsers
- Consider an S-Attributed grammar for constant expressions:
  - Each nonterminal has a single synthetic attribute: *val*
  - The annotated parse tree for **5 + 2 * 3** is shown below

Production	Semantic Rules
$E \rightarrow E^2 + T$	$E.val := E^2.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T^2 * F$	$T.val := T^2.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ **num**	$F.val :=$ **num**$.val$



Constructing Syntax Trees for Expressions

- A syntax tree is a condensed form of a parse tree
- A syntax tree can be used as an intermediate representation
- Each node is a structure with several fields
- To construct a syntax tree, we need …
  - *mknode*(*op, left, right*) creates a new node for a binary operator
    - *op* is a binary operator
    - *left* and *right* are pointers to the left and right subtrees
  - *idTable.lookup*(*name*) searches the identifier table for a given *name*
    - Returns a pointer to the found identifier symbol
    - Returns NULL if *name* is not found

S-Attributed Grammar for Syntax Trees

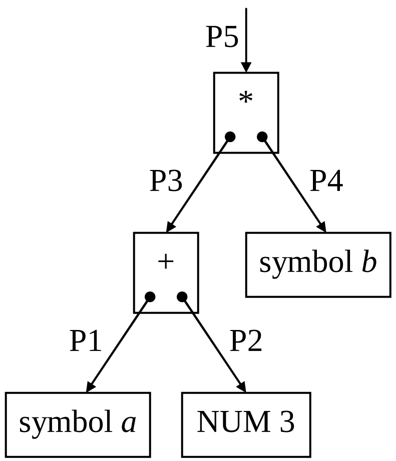
- An S-attributed grammar is used for constructing a syntax tree
- A synthetic attribute *ptr* is used with E, T, F and **num**
  - *ptr* is a pointer that points at the syntax generated for E, T,and F
  - *ptr* is also used to point at a literal symbol for the token **num**

Production	Semantic Rules	Yacc Notation
$E \rightarrow E^2 + T$	$E.ptr := mknode('+', E^2.ptr, T.ptr)$	`$$ = mknode('+', $1, $3);`
$E \rightarrow E^2 - T$	$E.ptr := mknode('-', E^2.ptr, T.ptr)$	`$$ = mknode('–', $1, $3);`
$E \rightarrow T$	$E.ptr := T.ptr$	`$$ = $1;`
$T \rightarrow T^2 * F$	$T.ptr := mknode('*', T^2.ptr, F.ptr)$	`$$ = mknode('*', $1, $3);`
$T \rightarrow T^2 / F$	$T.ptr := mknode('/', T^2.ptr, F.ptr)$	`$$ = mknode('/', $1, $3);`
$T \rightarrow F$	$T.ptr := F.ptr$	`$$ = $1;`
$F \rightarrow$ **(** E **)**	$F.ptr := E.ptr$	`$$ = $2;`
$F \rightarrow$ **id**	$F.ptr := idTable.lookup($**id**$.name)$	`$$ = idTable.lookup($1);`
$F \rightarrow$ **num**	$F.ptr :=$ **num**$.ptr$	`$$ = $1;`

Generation of a Syntax Tree by an LR Parser

- Synthesized attributes can be easily computed by an LR parser
  - An LR parser will have a **value stack** for storing synthesized attributes
  - The **value stack** is manipulated in parallel with the **parsing stack**
- Consider the generation of the syntax tree of: **(a + 3) * b**

Parsing Stack	Input	Action	Semantic Action	Value Stack
	(a + 3) * b $	shift (		
(	a + 3) * b $	shift id		?
(id	+ 3) * b $	reduce F → id	F.nptr := lookup(*a*)	? a
(F	+ 3) * b $	reduce T → F	T.nptr := F.nptr	? P1
(T	+ 3) * b $	reduce E → T	E.nptr := T.nptr	? P1
(E	+ 3) * b $	shift +		? P1
(E +	3) * b $	shift num		? P1 ?
(E + num	) * b $	reduce F → num	F.nptr := num.ptr	? P1 ? P2
(E + F	) * b $	reduce T → F	T.nptr := F.nptr	? P1 ? P2
(E + T	) * b $	reduce E → E^2 + T	E.nptr := mknode('+', E^2.nptr, T.nptr)	? P1 ? P2
(E	) * b $	shift)		? P3
(E)	* b $	reduce F → (E)	F.nptr := E.nptr	? P3 ?
F	* b $	reduce T → F	T.nptr := F.nptr	P3
T	* b $	shift *		P3
T *	b $	shift id		P3 ?
T * id	$	reduce F → id	F.nptr := lookup(*b*)	P3 ? b
T * F	$	reduce T → T^2 * F	T.nptr := mknode('*', T^2.nptr, F.nptr)	P3 ? P4
T	$	reduce E → T	E.nptr := T.nptr	P5
E	$	Accept		P5



L-Attributed Grammars

- Consider a typical production of the form: $A \to X_1\ X_2\ ...\ X_n$
- An attribute grammar is L-attributed if and only if:
  - Each inherited attribute of a right-hand-side symbol X_j depends only on inherited attributes of A and arbitrary attributes of the symbols $X_1, \dots, X_{j-1}$
  - Each synthetic attribute of A depends only on its inherited attributes and arbitrary attributes of the right-hand side symbols: $X_1\ X_2\ ...\ X_n$
- When Evaluating the attributes of an L-attributed production:
  - Evaluate the inherited attributes of A (left-hand-side)
  - Evaluate the inherited then the synthesized attributes of X_j from left to right
  - Evaluate the synthesized attribute of A
- If the underlying CFG is LL and L-attributed, we can evaluate the attributes in one pass by an LL Parser
- Every S-attributed grammar is also L-attributed

L-Attributed Grammar Evaluation

- L-attributed grammars are well-suited for LL-based evaluation
- Consider the prediction of production: $A \rightarrow X\ Y$
  - Evaluate and Push Inherited attributes of *A*: … Inh(*A*)
  - Evaluate and Push Inherited attributes of *X*: … Inh(*A*) Inh(*X*)
  - Evaluate and Push Synthetic attributes of *X* after parsing *X*:

    … Inh(*A*) Inh(*X*) Syn(*X*)
  - Evaluate and Push Inherited attributes of *Y*:

    … Inh(*A*) Inh(*X*) Syn(*X*) Inh(*Y*)
  - Evaluate and Push Synthetic attributes of *Y* after parsing *Y*:

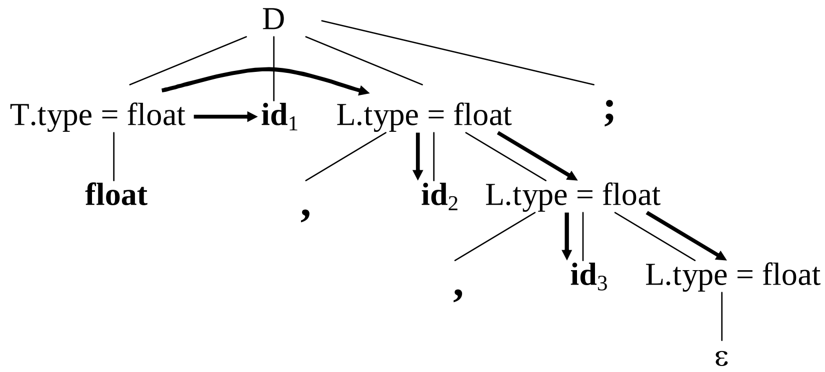
    … Inh(*A*) Inh(*X*) Syn(*X*) Inh(*Y*) Syn(*Y*)
  - Pop attributes of *X* and *Y* and push Synthetic attributes of *A*:

    … Inh(*A*) Syn(*A*)
- Attribute values are at **known locations** relative to *stacktop*

Example of an L-Attributed Grammar

- A C-like declaration generated by the non-terminal *D* consists of
  - Keyword **int** or **float**, followed by a list of identifiers
- The non-terminal *T* has a **synthesized attribute** *type*
- The non-terminal *L* has an **inherited attribute** *type*
- The function *enter* creates a new symbol entry in a symbol table

Production	Semantic Rules
$D \rightarrow T$ **id** L **;**	*enter*(**id**.*name*, *T*.*type*) *L*.*type* := *T*.*type*
$T \rightarrow$ **int**	*T*.*type* := INT_TYPE
$T \rightarrow$ **float**	*T*.*type* := FLOAT_TYPE
$L \rightarrow$ **,** **id** L^2	enter(**id**.*name*, L.*type*) L^2.*type* := L.*type*
$L \rightarrow \varepsilon$	

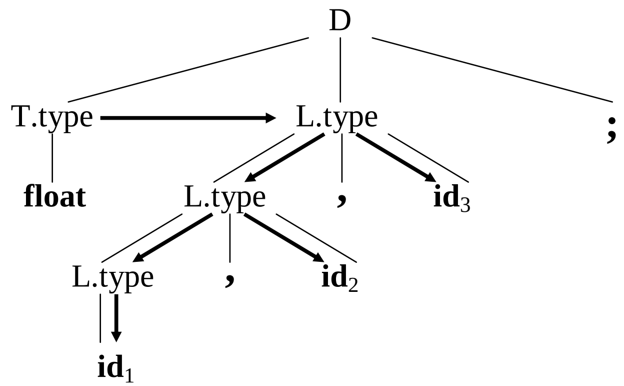


Parse tree for: **float id , id , id ;**

Inheriting Attributes During LR Parsing

- Some L-attributed grammars can be used with LR parsers
- Consider the L-attributed grammar for C-like declarations:
  - The grammar is LR(1) but not LL(1) because of left-recursion.
  - The non-terminal *L* has an inherited attribute *type* defined by *L.type* := *T.type*
  - The attribute *T.type* will be on the value stack when reduction to *L* takes place
  - The synthetic attribute *T.type* can be used anywhere *L.type* is accessed

Production	**Semantic Rules**
$D \rightarrow T\ L$ **;**	*L.type* := *T.type*
$T \rightarrow$ **int**	T.type := int
$T \rightarrow$ **float**	T.type := float
$L \rightarrow L^2$ **, id**	enter(**id**.name, L.type)
$L \rightarrow$ **id**	L^2.type := L.type enter(**id**.name, L.type)



Parse tree for: **float id , id , id ;**

Inherited Attributes in Yacc: $0, $-1, $-2, …

- Yacc allows the inheritance of previously computed attributes:
  - Access to inherited attributes is done via $0, $-1, $-2, etc.
  - These attributes are stacked *below* the attributes of the current production.
  - Inherited attributes can be very useful, but can also be a source of bugs.
  - In the example shown below, $0 refers to the attribute of *T* stacked below the attributes of the symbols of the *L* production.
  - If the first production of *L* is right-recursive, the use of $0 will not work.

Production	*Yacc Actions*
D → T L **;**	
T → **int**	$$ = 1; /* int */
T → **float**	$$ = 2; /* float */
L → L **, id**	enter($3, $0);
L → **id**	enter($1, $0)

Stack	*Input*	*Action*	*Semantic Action*	*Value Stack*
	int a , b ; $	shift int		
int	a , b ; $	reduce T → int	$$ = 1	?
T	a , b ; $	shift id		1
T id	, b ; $	reduce L → id	enter($1, $0)	1 a
T L	, b ; $	shift ,		1 ?
T L ,	b ; $	shift id		1 ? ?
T L , id	; $	reduce L → L , id	enter($3, $0)	1 ? ? b
T L	; $	shift ;		1 ?
T L ;	$	reduce D → T L ;		1 ? ?
D	$	Accept		?

Replacing Inherited Attributes by Synthesized Ones

- It is sometimes possible to avoid the use of inherited attributes
- This requires changing the underlying grammar
- Consider a Pascal-like declaration:
  - The first grammar uses an inherited attribute *type* for *L*
  - However, the first grammar is NOT L-attributed because *L.type* inherits the attribute of a right-sibling *T*
  - The second grammar is S-attributed. It uses synthetic attributes only.

Production	*Semantic Rules*
D → L **:** T **;**	L.type := T.type
L → L^2 **, id**	enter(**id**.name, L.type) L^2.type := L.type
L → **id**	enter(**id**.name, L.type)
T → **integer**	T.type := integer
T → **real**	T.type := real

Production	*Semantic Rules*
D → **id** L	enter(**id**.name, L.type)
L → **, id** L^2	enter(**id**.name, L^2.type) L.type := L^2.type
L → **:** T **;**	L.type := T.type
T → **integer**	T.type := integer
T → **real**	T.type := real