Semantic Analysis

a.y. 2022-2023

Semantic Analysis

- Compute additional information once the syntactic structure is known
- Information that is beyond the capabilities of context-free grammars
- Typically
  - Populate symbol tables after declarations (e.g., which is the type of declared variable?)
  - Perform type inference and type checking on expressions and statements
- Two categories of analysis:
  - Analysis required to establish correctness
  - Analysis to enhance efficiency of the translated program

Semantic Analysis

- Handiest method to describe semantic analysis is to identify properties (**attributes**) of grammar symbols and to write rules (**semantic rules**) to describe how the computation of those properties is related to the grammar productions
- This is what we call **attributed grammar**, or **syntax-directed definition**
- Instead of the derivation tree, a better basis for semantic computations is the **abstract syntax tree** (a compact representation of the derivation tree). However, the abstract syntax tree can itself be defined by an appropriate instance of semantic analysis

Semantic Analysis
WHAT NEXT

- Syntax-Directed Definitions
- Techniques for the computation of the specified attributes

Syntax-Directed Definitions

- Syntax-directed definitions (SDDs) are context-free grammars enriched with attributes and rules to compute them
  - **Attributes**: Associated with grammar symbols; can be numbers, types, references to the symbol table, location of a variable in memory, object code of a procedure, etc.
  - **Semantic rules**: Associated with productions; typically induce the computation of some attributes as functions of the attributes of other symbols of the production
- Used to perform semantic analysis and more

Attributes

Attributes of non-terminals are classified as

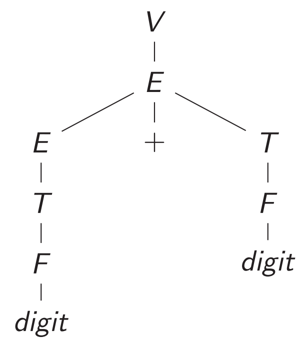
- **Synthesized:** attributes of the driver defined as a function of the attributes of symbols in the production
- **Inherited:** attributes of non-terminals in the body defined as a function of the attributes of symbols in the production

Attributes of terminals can only be synthesized: They are supplied by the lexical analyzer, there is no rule to compute them

SDD for arithmetic expressions

EXAMPLE

Consider the following parse tree for the SLR(1) grammar of arithmetic expressions



If one *digit* has lexical value 3 and the other *digit* has lexical value 4, then the value of the expression should be 7

SDD for arithmetic expressions

EXAMPLE

$$V \to E \qquad \{V.val = E.val\}$$

$$E \to E_1 + T \qquad \{E.val = E_1.val + T.val\}$$

$$E \to T \qquad \{E.val = T.val\}$$

$$T \to T_1 * F \qquad \{T.val = T_1.val * F.val\}$$

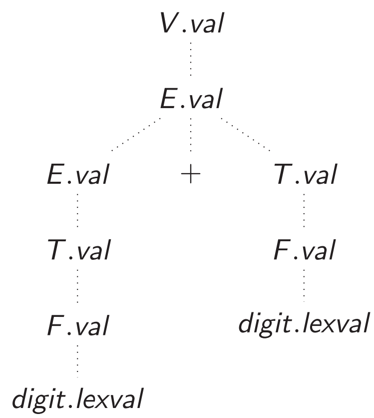
$$T \to F \qquad \{T.val = F.val\}$$

$$F \to (E) \qquad \{F.val = E.val\}$$

$$F \to digit \qquad \{F.val = digit.lexval\}$$

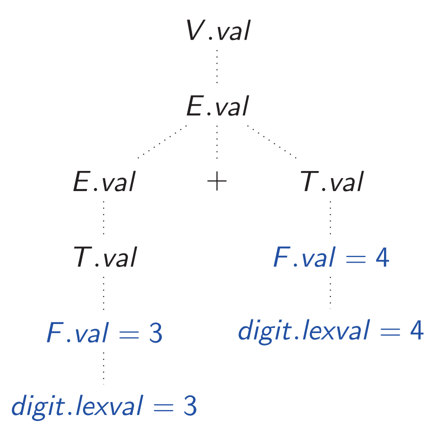
Evaluation of SDDs

Annotated parse tree: Parse tree enriched with the values of attributes at its nodes



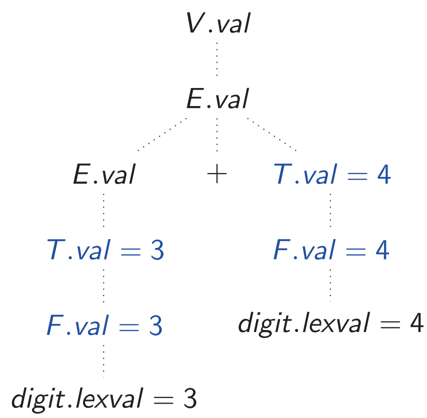
Evaluation of SDDs

$$F \to digit \qquad \{F.val = digit.lexval\}$$



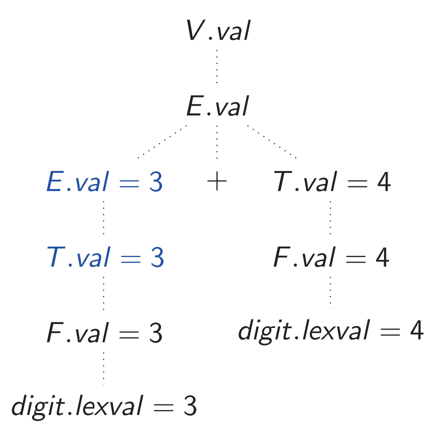
Evaluation of SDDs

$$T \to F \qquad \{T.val = F.val\}$$



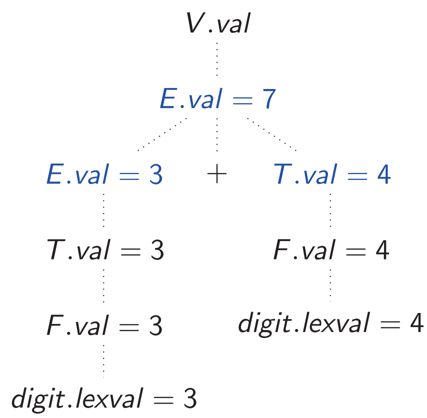
Evaluation of SDDs

$$E \to T \qquad \{E.val = T.val\}$$



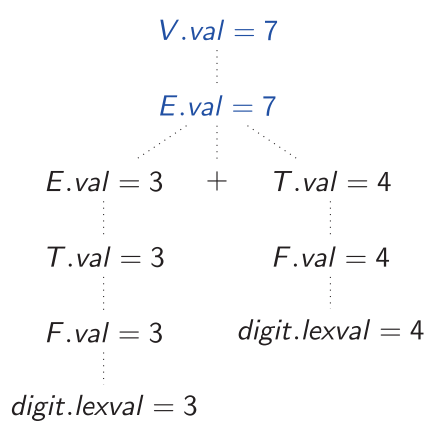
Evaluation of SDDs

$$E \to E_1 + T \qquad \{E.val = E_1.val + T.val\}$$



Evaluation of SDDs

$$V \to E \qquad \{V.val = E.val\}$$



Evaluation order of SDDs

Define a **dependency graph** for the SDD:

- Set a node of the dependency graph for each attribute associated with each parse tree node
- For each attribute $X.x$ used to define the attribute $Y.y$, set an edge from the node for $X.x$ to the node for $Y.y$.
  The edge means "$X.x$ **is needed to define** $Y.y$"

The SDD can be evaluated after any **topological sort** of the dependency graph.

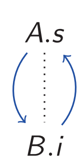
To get a topological sort, number the nodes $N_1, \ldots, N_k$ of the dependency graph in such a way that if there is an edge from N_i to N_j then $i < j$.

Evaluation of SDDs

When the SDD has both syntesized and inherited attributes there is no guarantee that a topological sort (and hence an evaluation order) exists

There might be a cycle in the dependency graph, hence no topological sort:

$$A \rightarrow B \qquad \{A.s = B.i;\ B.i = A.s + 7\}$$



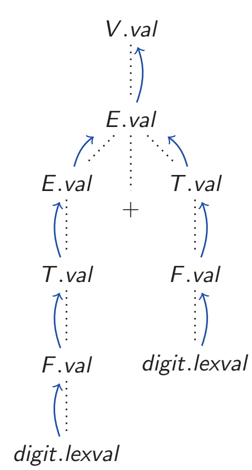
Evaluation of SDDs

Classes of SDDs for which the existence of a topological sort of the dependency graph is guaranteed:

- **S-attributed SDDs**: Only synthesized attributes, can just use post-order to evaluate
- **L-attributed SDDs**: Either synthesized attributes, or inherited attributes such that
  - For each production $A \to X_1 \dots X_n$
  - The definition of each $X_j.i$ uses at most
    - Inherited attributes of A
    - Inherited or synthesized attributes of the left siblings $X_1, \dots, X_{j-1}$

Example: SLR(1) Arithmetic Expressions

DEPENDENCY GRAPH



Example: SLR(1) Arithmetic Expressions
EVALUATION OF SDD

- Computing a topological sort of the nodes is superfluous
- A **post-order visit** of the parse tree is enough to evaluate the SDD

This is always the case for S-attributed SDDs

```
post-order(N) {
    foreach child C of N do post-order(C);
    evaluate attributes of N;
}
```


Example: SLR(1) Arithmetic Expressions
EVALUATION OF SDD

```
procedure expEval(N)
    foreach child C of N do
        expEval(C) ;
    if N is a Fnode then
        if N has only one child then
            assign to N.val the attribute digit.lexval of its child
        else
            assign to N.val the attribute of its second child
    else if N is a Enode then
        if N has only one child then
            assign to N.val the attribute of its child
        else
            assign to N.val the sum of the attributes of its first
            and third children
    ...
```

Example: Typed declarations

Sometimes synthesized attributes are not appropriate

Consider the following grammar for variable declaration:

$$\begin{array}{lcl} D & \rightarrow & TL \\ T & \rightarrow & int \\ T & \rightarrow & float \\ L & \rightarrow & L, id \\ L & \rightarrow & id \end{array}$$

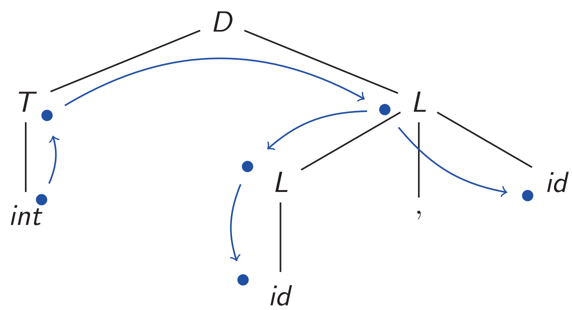
Design semantic actions to assign the declared type (either "integer" or "float) to the attributes $id.type$

Formal Languages and Compilers Paola Quaglia, 2022

Example: Typed declarations

$$\begin{array}{ll} D \rightarrow TL & \{L.type = T.type\} \\ T \rightarrow int & \{T.type = integer\} \\ T \rightarrow float & \{T.type = float\} \\ L \rightarrow L_1, id & \{L_1.type = L.type; id.type = L.type\} \\ L \rightarrow id & \{id.type = L.type\} \end{array}$$

Example: Typed declarations



One single inherited attribute and what about evaluation now?

Either pre-compute a topological sort, or apply an ad hoc mix of in-order and preorder

Example: Typed declarations

EVALUATION OF SDD

```
procedure typeEval(N)
    if N is a Dnode then
        typeEval(Tnode child of N) ;
        assign the type of the Tnode child to the Lnode child ;
        typeEval(Lnode child of N);
    else if N is a Lnode then
        assign N.type to the IDnode child ;
        if N has a Lnode child then
            assign N.type to the Lnode child ;
            typeEval (Lnode child of N) ;
    else if N is a Tnode then
        assign integer to N.type if child of N is int, assign float
        to N.type otherwise
```

Example: LL(1) Arithmetic Expressions

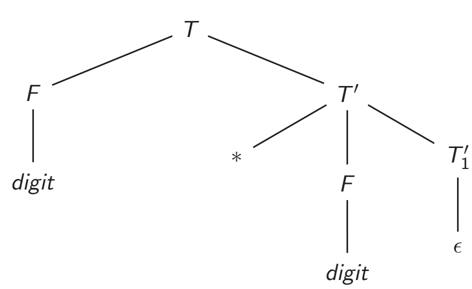
Different grammars pose different challenges in designing SDDs

Consider the LL(1) grammar for arithmetic expressions:

$$\begin{array}{lcl} V & \rightarrow & E \\ E & \rightarrow & TE' \\ E' & \rightarrow & +TE' \mid \epsilon \\ T & \rightarrow & FT' \\ T' & \rightarrow & *FT' \mid \epsilon \\ F & \rightarrow & (E) \mid digit \end{array}$$

Example: LL(1) Arithmetic Expressions

The parse tree for $3 * 5$ has the subtree

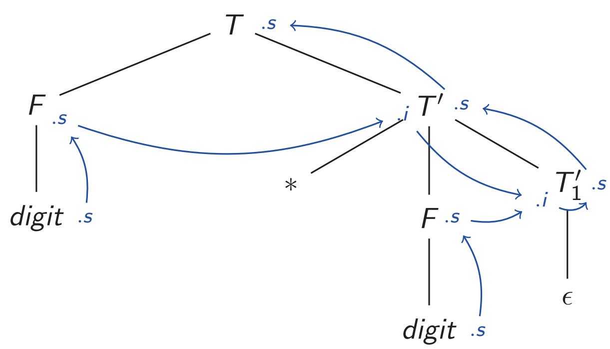


Here one *digit* is on the left subtree of the root, while the operator $*$ and the other digit are on the right subtree of the root

How can we get the expected $T.val = 15$?

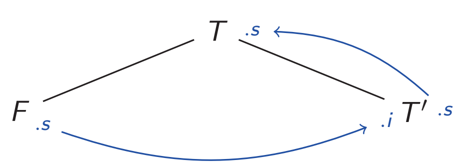
Example: LL(1) Arithmetic Expressions

Both inherited and synthesized attributes together



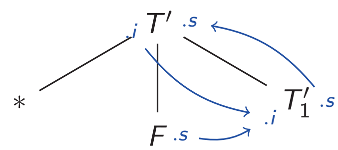
Where every *.s* stands for a syntesized attribute of the node, and every *i* stands for an inherited attribute of the node

Example: LL(1) Arithmetic Expressions



$T \to FT' \qquad \{T.s = T'.s;\ T'.i = F.s\}$

Example: LL(1) Arithmetic Expressions



$T' \to *FT'_1 \qquad \{T'.s = T'_1.s;\ T'_1.i = T'.i * F.s\}$

Formal Languages and Compilers Paola Quaglia, 2022

Example: LL(1) Arithmetic Expressions

L-attributed SDD for arithmetic expressions

$V \to E$	$\{V.s = E.s\}$
$E \to TE'$	$\{E.s = E'.s;\ E'.i = T.s\}$
$E' \to +TE'_1$	$\{E'.s = E'_1.s;\ E'_1.i = E'.i + T.s\}$
$E' \to \epsilon$	$\{E'.s = E'.i\}$
$T \to FT'$	$\{T.s = T'.s;\ T'.i = F.s\}$
$T' \to *FT'_1$	$\{T'.s = T'_1.s;\ T'_1.i = T'.i * F.s\}$
$T' \to \epsilon$	$\{T'.s = T'.i\}$
$F \to (E)$	$\{F.s = E.s\}$
$F \to digit$	$\{F.s = digit.lexval\}$

Evaluation during bottom-up parsing

The challenge is to implement the translation during parsing (vs obtaining the parse tree, then annotating it, then evaluating the annotated parse tree)

By far the simplest implementation occurs when the grammar can be parsed by the shift/reduce algorithm and the underlying SDD is S-attributed

Together with the state stack *stSt* and the symbol stack *symSt*, use a semantic stack *semSt* to keep (records of) attributes

Execute the code associated with $A \to \beta$ when the production is reduced

The needed attributes of the symbols on top of *symSt* are found at the corresponding positions on top of *semSt*

Evaluation during bottom-up parsing

EXAMPLE

SDDs are at the basis of syntax-directed translation schemes, where the computation of attributes goes together with fragments of code that use them

For example, if we read the "=" sign as an assignment, then

$$
\begin{array}{ll}
V \to E & \{print(E.val)\} \\
E \to E_1 + T & \{E.val = E_1.val + T.val\} \\
E \to T & \{E.val = T.val\} \\
T \to T_1 * F & \{T.val = T_1.val * F.val\} \\
T \to F & \{T.val = F.val\} \\
F \to (E) & \{F.val = E.val\} \\
F \to digit & \{F.val = digit.lexval\}
\end{array}
$$

Is a syntax-directed translation

Evaluation during bottom-up parsing

EXAMPLE

Run the shift/reduce algorithm on input "$digit + digit$" for the LALR(1) grammar of arithmetic expressions

Input	Stack				
$digit + digit\$$	$stSt$	0			
	$symSt$				
	$semSt$				

Input	Stack				
$\underline{digit}+digit\$$	$stSt$	0	1		
	$symSt$		digit		
	$semSt$		3	push $digit.lexval$	

Input	Stack			
$\underline{digit}+digit\$$	$stSt$	0	6	
	$symSt$		F	reduce $F \to digit$
	$semSt$		3	pop $digit.lexval$; push $F.val$

Translation during bottom-up parsing

EXAMPLE

Input	Stack			
$\underline{digit}+digit\$$	$stSt$	0	5	
	$symSt$		T	reduce $T \to F$
	$semSt$		3	pop $F.val$; push $T.val$

Input	Stack			
$\underline{digit}+digit\$$	$stSt$	0	4	
	$symSt$		E	reduce $E \to T$
	$semSt$		3	pop $T.val$; push $E.val$

Input	Stack				
$\underline{digit+}digit\$$	$stSt$	0	4	9	
	$symSt$		E	+	
	$semSt$		3	0	push a dummy number to keep stacks aligned

Input	Stack					
$\underline{digit + digit}\$$	$stSt$	0	4	9	1	
	$symSt$		E	+	digit	
	$semSt$		3	0	4	push $digit.lexval$

Translation during bottom-up parsing

EXAMPLE

$\underline{digit + digit}\$$	*stSt*	0	4	9	6	
	symSt	E	$+$	F		reduce $F \to digit$
	semSt	3	0	4		pop $digit.lexval$; push $F.val$

$\underline{digit + digit}\$$	*stSt*	0	4	9	12	
	symSt	E	$+$	T		reduce $T \to F$
	semSt	3	0	4		pop $F.val$; push $T.val$

$\underline{digit + digit}\$$	*stSt*	0	4	
	symSt	E		reduce $E \to E + T$
	semSt	7		pop $T.val$; pop; pop $E_1.val$;
				push $E.val = E_1.val + T.val$

$\underline{digit + digit}\$$	*stSt*	0	1	
	symSt	V		reduce $V \to E$; Accept
	semSt	7		$print(7)$

Extended example

FROM STRINGS TO NUMBERS: G0

Goal: translate strings of digits to their decimal number values

$$
\begin{array}{ll}
S \to Digits & \{print(Digits.v)\} \\
Digits \to Digits_1\ d & \{Digits.v = Digits_1.v * 10 + d.lexval\} \\
Digits \to d & \{Digits.v = d.lexval\}
\end{array}
$$

The grammar is LALR(1), the SDD is S-attributed, attributes can be easily evaluated

Extended example

FROM STRINGS TO NUMBERS

Goal: if the string of digits is prefixed by the terminal o then translate it to its octal number value, otherwise translate it to its decimal number value

Extended example

FROM STRINGS TO NUMBERS: G1

First attempt: simply add one production to G0 to generate strings of digits prefixed by o

$$\begin{array}{l} S \rightarrow Num \\ Num \rightarrow o\ Digits \mid Digits \\ Digits \rightarrow Digits\ d \mid d \end{array}$$

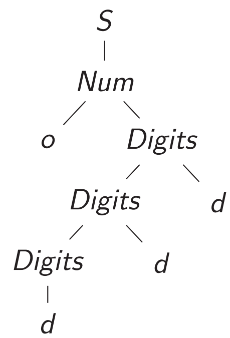
The grammar is LALR(1)

Can we compute number values using synthesized attributes?

Extended example

FROM STRINGS TO NUMBERS: G1

$S \rightarrow Num$
$Num \rightarrow o\ Digits \mid Digits$
$Digits \rightarrow Digits\ d \mid d$



The information required to get the translation right is not available from below, cannot be synthesized

Bad luck, change the grammar

Extended example

FROM STRINGS TO NUMBERS: G2

Second attempt: clone productions to separate the management of octal numbers from that of decimal numbers

$S \rightarrow Num$	$\{print(Num.v)\}$
$Num \rightarrow o\ O$	$\{Num.v = O.v\}$
$Num \rightarrow D$	$\{Num.v = D.v\}$
$O \rightarrow O_1\ d$	$\{O.v = O_1.v * 8 + d.lexval\}$
$O \rightarrow d$	$\{O.v = d.lexval\}$
$D \rightarrow D_1\ d$	$\{D.v = D_1.v * 10 + d.lexval\}$
$D \rightarrow d$	$\{D.v = d.lexval\}$

LALR(1), good for synthesising, but redundant

Extended example

FROM STRINGS TO NUMBERS: G3

Third attempt: introduce extra unit productions to force semantic actions

$S \to Num$	$\{print(Num.v)\}$
$Num \to Octal\ Digits$	$\{Num.v = Digits.v\}$
$Num \to Decimal\ Digits$	$\{Num.v = Digits.v\}$
$Octal \to o$	$\{base = 8\}$
$Decimal \to \epsilon$	$\{base = 10\}$
$Digits \to Digits_1\ d$	$\{D.v = D_1.v * base + d.lexval\}$
$Digits \to d$	$\{Digits.v = d.lexval\}$

LALR(1), but uses the global variable *base*

Extended example

FROM STRINGS TO NUMBERS: G4

Fourth attempt: let the appropriate base be captured by a synthesized attribute

$S \to D$	$\{print(D.v)\}$
$D \to D\ d$	$\{D.v = D_1.v * D_1.base + d.lexval;$ $D.base = D_1.base\}$
$D \to B\ d$	$\{D.v = d.lexval;$ $D.base = B.val\}$
$B \to o$	$\{B.val = 8\}$
$B \to \epsilon$	$\{B.val = 10\}$

Another extended example

CONVERSION FROM INTEGER TO FLOAT

Take the following grammar

$$E \rightarrow E \; dop \; T \mid T$$
$$T \rightarrow num \mid num.num$$

Goal: add semantic actions to compute the value of the expression so that:

- If the expression contains only integer numbers (terminal *num*) then *dop* is translated to the integer division operator *div*
- If the expression contains at least a real number (terminal *num.num*) then the whole expression is converted to a floating point expression, with integers promoted to floating-point numbers. For example 5 *dop* 2 *dop* 2.0 is converted to *float*(5) / *float*(2) / 2.0

Another extended example

CONVERSION FROM INTEGER TO FLOAT

We need to know whether the parse tree has any *num.num* leaf, and we can compute the value of the expression only when we know whether the operands have to be promoted to floating-point or not. Then:

- We use a boolean attribute *float* which flows from the leaves to the root (synthesized attribute)

$$\begin{array}{ll} E \rightarrow E_1 \; dop \; T & E.float = E_1.float \;\; OR \;\; T.float; \ldots \\ E \rightarrow T & E.float = T.float; \ldots \\ T \rightarrow num & T.float = false; \ldots \\ T \rightarrow num.num & T.float = true; \ldots \end{array}$$

Another extended example

CONVERSION FROM INTEGER TO FLOAT

- If the tree has any *num.num* leaf, then the *float* attribute of the root is *true*. Once we know the *float* attribute of the root, we can decide the type of the whole expression and let it flow towards the leaves through the attribute *type* (inherited attribute)
- To be sure that we decide the type of the expression just at the root of the parse tree (rather than at some E-node which cannot access all the leaves of the tree), we add the extra production

  $$S \to E$$

  to distinguish to E-node playing the root. The *type* attribute for E is set at this extra production

Another extended example

CONVERSION FROM INTEGER TO FLOAT

$S \to E$	$E.type = IF\ E.float\ THEN\ float\ ELSE\ int; \ldots$
$E \to E_1\ dop\ T$	$E_1.type = E.type;\ T.type = E.type; \ldots$
$E \to T$	$T.type = E.type; \ldots$
$T \to num$	$\ldots$
$T \to num.num$	$\ldots$

Another extended example

CONVERSION FROM INTEGER TO FLOAT

- The value of the expression, which depends on its *type*, is computed in the attribute *val* (synthesized attribute)

$S \rightarrow E$	$S.val = E.val; \ldots$
$E \rightarrow E_1 \; dop \; T$	$E.val = IF \;\; E.type = int$ $THEN \;\; E_1.val \;\; div \;\; T.val$ $ELSE \;\; E_1.val / T.val; \ldots$
$E \rightarrow T$	$E.val = T.val; \ldots$
$T \rightarrow num$	$T.val = IF \;\; T.type = int$ $THEN \;\; num.lexval$ $ELSE \;\; float(num.lexval); \ldots$
$T \rightarrow num.num$	$T.val = num.num.lexval; \ldots$

Another extended example

CONVERSION FROM INTEGER TO FLOAT

And now:

Collect all the semantic actions and get the SDD

Get the annotated parse tree

Evaluate

Abstract syntax trees

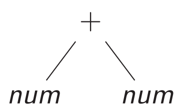
Abstract syntax trees

Compact representations of parse trees, often taken as intermediate representation

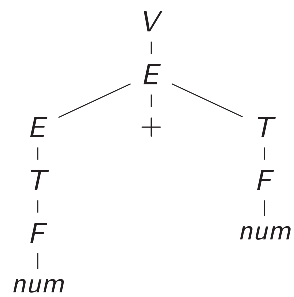
No general rule to define them, it depends on the grammar at hand and on the implementation choice

Surely they must contain all the information needed to carry on the analysis

Abstract syntax trees



instead of



Abstract syntax trees

Provide an S-attributed grammar so that the abstract syntax tree can be constructed while parsing

Assumptions:

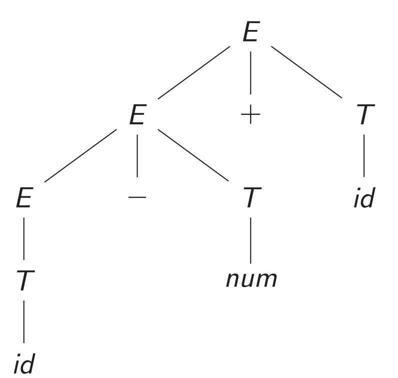
- Function $newLeaf(label, val)$ creates a leaf object with two fields: the label of the node and its value
- Function $newNode(label, c_1, \ldots, c_k)$ creates an internal node with k children: $label$ is the label of the node, and $c_1, \ldots, c_k$ are references to the children nodes

Abstract syntax trees
ARITHMETIC EXPRESSIONS: LALR GRAMMAR

$E \rightarrow E_1 + T$	$\{E.node = newNode(\text{'}+\text{'}, E_1.node, T.node)\}$
$E \rightarrow E_1 - T$	$\{E.node = newNode(\text{'}-\text{'}, E_1.node, T.node)\}$
$E \rightarrow T$	$\{E.node = T.node\}$
$T \rightarrow (E)$	$\{T.node = E.node\}$
$T \rightarrow id$	$\{T.node = newLeaf(id, id.entry)\}$
$T \rightarrow num$	$\{T.node = newLeaf(num, num.lexval)\}$

Abstract syntax trees
ARITHMETIC EXPRESSIONS: LALR GRAMMAR

Parse $\boldsymbol{id} - \boldsymbol{num} + \boldsymbol{id}$ resulting from lexing $\boldsymbol{a} - 4 + \boldsymbol{c}$



Reductions:

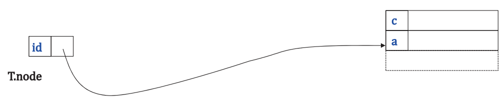
1. $T \rightarrow id$
2. $E \rightarrow T$
3. $T \rightarrow num$
4. $E \rightarrow E - T$
5. $T \rightarrow id$
6. $E \rightarrow E + T$

Abstract syntax trees

ARITHMETIC EXPRESSIONS: LALR GRAMMAR

First reduction:

$$T \to id \qquad \{T.node = newLeaf(id, id.entry)\}$$

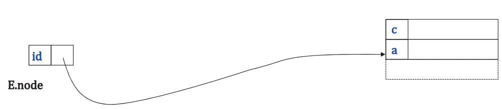


Abstract syntax trees

ARITHMETIC EXPRESSIONS: LALR GRAMMAR

Second reduction:

$$E \to T \qquad \{E.node = T.node\}$$

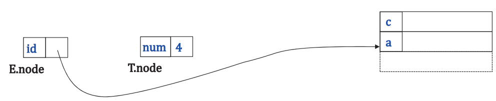


Abstract syntax trees

ARITHMETIC EXPRESSIONS: LALR GRAMMAR

Third reduction:

$$T \to num \qquad\qquad \{T.node = newLeaf(num, num.lexval)\}$$

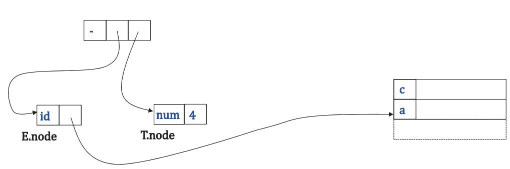


Abstract syntax trees

ARITHMETIC EXPRESSIONS: LALR GRAMMAR

Fourth reduction:

$$E \to E_1 - T \qquad\qquad \{E.node = newNode('-', E_1.node, T.node)\}$$

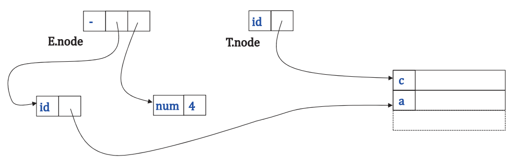


Abstract syntax trees

ARITHMETIC EXPRESSIONS: LALR GRAMMAR

Fifth reduction:

$$T \rightarrow id \qquad \{T.node = newLeaf(id, id.entry)\}$$

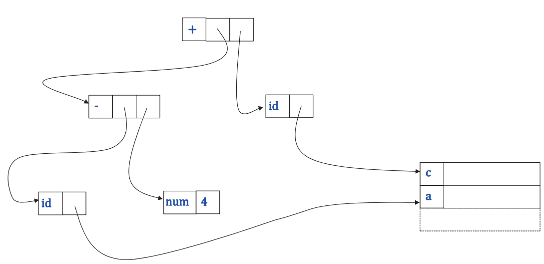


Abstract syntax trees

ARITHMETIC EXPRESSIONS: LALR GRAMMAR

Last reduction:

$$E \rightarrow E_1 + T \qquad \{E.node = newNode('+', E_1.node, T.node)\}$$



Abstract syntax trees

ARITHMETIC EXPRESSIONS: LL GRAMMAR

$$
\begin{aligned}
&E \to TE' \\
&E' \to +TE'_1 \\
&E' \to -TE'_1 \\
&E' \to \epsilon \\
&T \to (E) \\
&T \to id \\
&T \to num
\end{aligned}
$$

Abstract syntax trees

ARITHMETIC EXPRESSIONS: LL GRAMMAR

We use synthesized and inherited attributes after a strategy similar to the case of the computation of the value of the expression

This time, rather than values, we pass around references to nodes

We get the annotated parse tree

Then we define the dependency graph and find a topological sort of its nodes

Last, evaluation constructs the abstract syntax tree

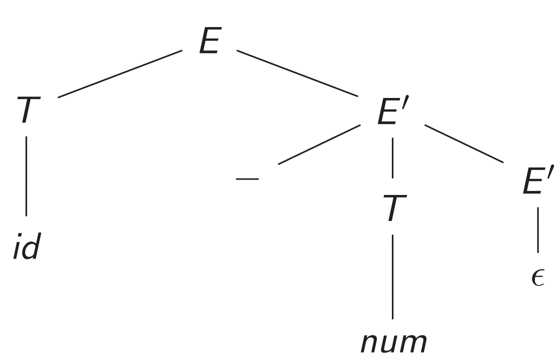
Abstract syntax trees
ARITHMETIC EXPRESSIONS: LL GRAMMAR

$E \to TE'$	$\{E.node = E'.node;$ $E'.i = T.node\}$
$E' \to +TE'_1$	$\{E'.node = E'_1.node;$ $E'_1.i = newNode(\text{'}+\text{'}, E'.i, T.node)\}$
$E' \to -TE'_1$	$\{E'.node = E'_1.node;$ $E'_1.i = newNode(\text{'}-\text{'}, E'.i, T.node)\}$
$E' \to \epsilon$	$\{E'.node = E'.i\}$
$T \to (E)$	$\{T.node = E.node\}$
$T \to id$	$\{T.node = newLeaf(id, id.entry)\}$
$T \to num$	$\{T.node = newLeaf(num, num.lexval)\}$

Abstract syntax trees
ARITHMETIC EXPRESSIONS: LL GRAMMAR

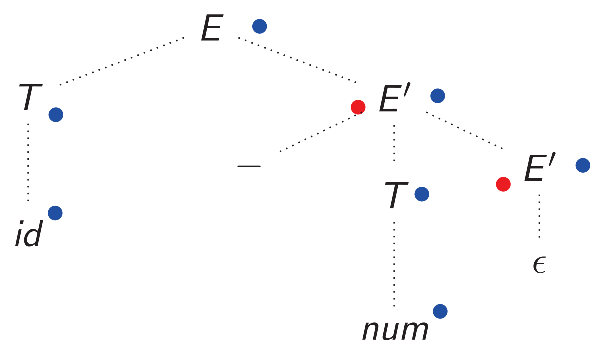
Parse ***id*** − ***num*** resulting from lexing ***a*** − 4

Parse tree



Abstract syntax trees
ARITHMETIC EXPRESSIONS: LL GRAMMAR

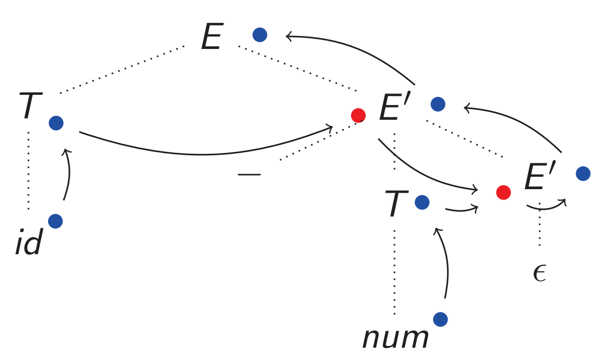
Annotated parse tree



Where ● stands for the inherited attribute of the tree node and ● stands for its synthesized attribute

Abstract syntax trees
ARITHMETIC EXPRESSIONS: LL GRAMMAR

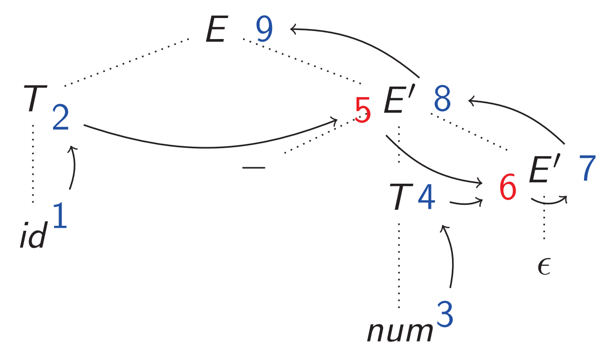
Dependency graph



Abstract syntax trees

ARITHMETIC EXPRESSIONS: LL GRAMMAR

Topological sort



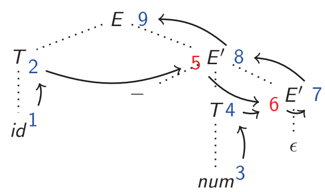
Formal Languages and Compilers Paola Quaglia, 2022

Abstract syntax trees

ARITHMETIC EXPRESSIONS: LL GRAMMAR

Abstract syntax trees

ARITHMETIC EXPRESSIONS: LL GRAMMAR



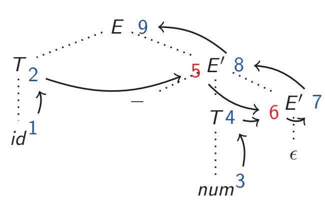
Evaluation:

1: *id.entry* (reference to the entry for *a* in the symbol table)

2: $T.node = newLeaf(id, 1)$

Abstract syntax trees

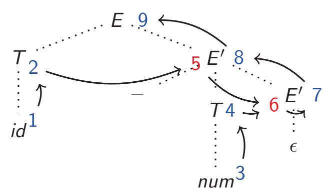
ARITHMETIC EXPRESSIONS: LL GRAMMAR



3: 4 (by $num.lexval = 4$)

4: $T.node = newLeaf(num, 3)$

Abstract syntax trees
ARITHMETIC EXPRESSIONS: LL GRAMMAR

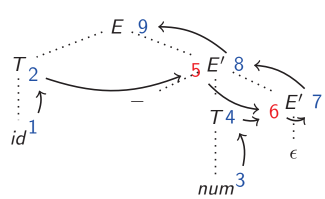


5: $E'.i = 2$

6: $E'.i = newNode(`-', 5, 4)$

Formal Languages and Compilers Paola Quaglia, 2022

Abstract syntax trees
ARITHMETIC EXPRESSIONS: LL GRAMMAR



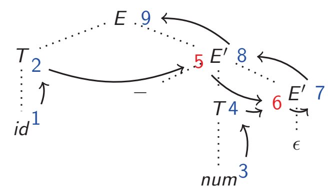
7: reference to $Node(`-', 5, 4)$
   By $E'.node = E'.i$ *for* $E' \to \epsilon$

8: reference to $Node(`-', 5, 4)$
   By $E'.node = E'_1.node$ *for* $E' \to -TE'_1$

9: reference to $Node(`-', 5, 4)$
   By $E.node = E'.node$ *for* $E \to TE'$

Abstract syntax trees

ARITHMETIC EXPRESSIONS: LL GRAMMAR



In overall:

$E.node$ is a reference to $Node('-', 5, 4)$

Where

- 5: reference to $Leaf(id, \text{entry for } a)$
- 4: reference to $Leaf(num, 4)$