Compiler Design

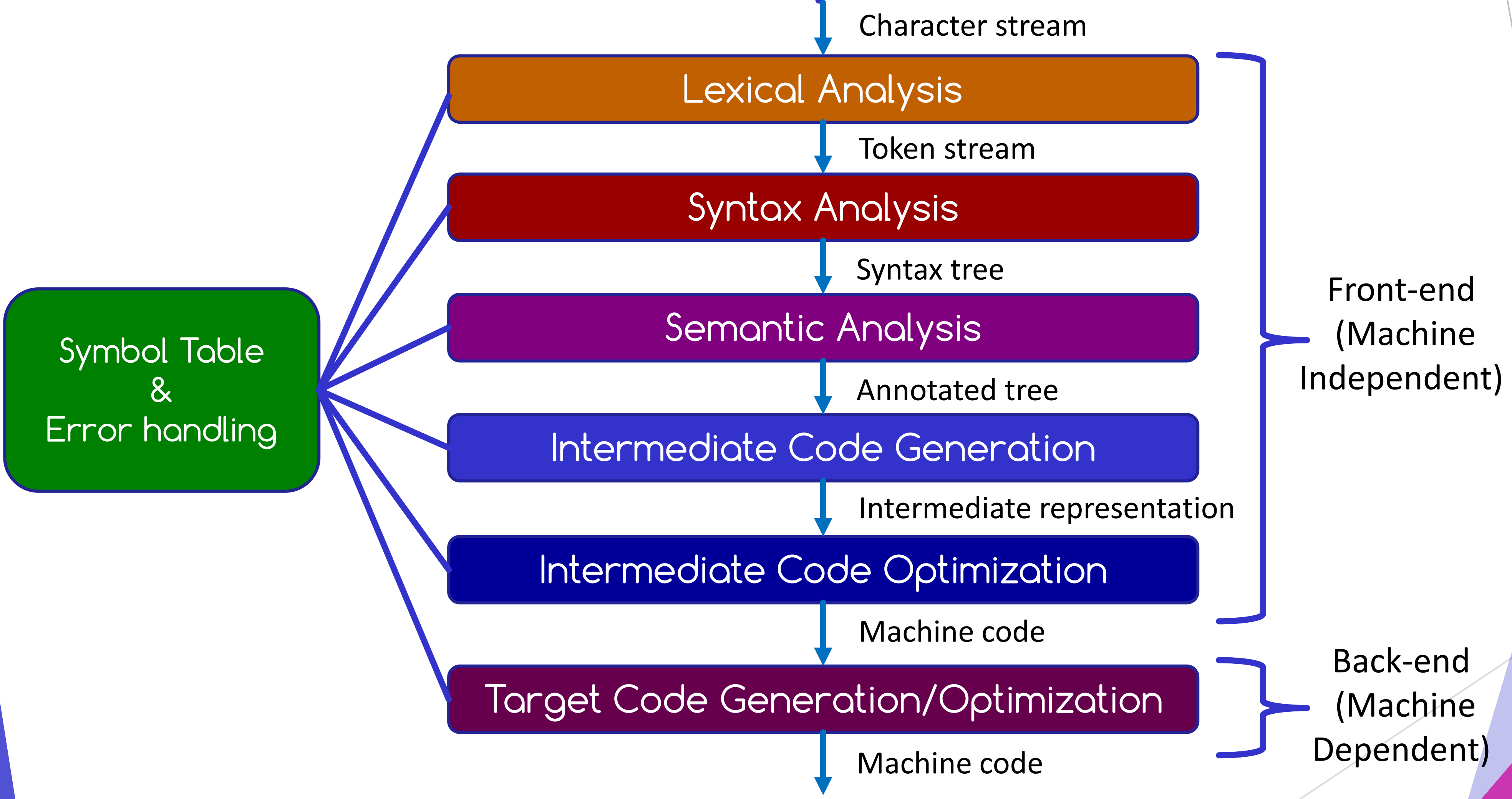
Lecture 9: Semantic Analysis

Sahar Selim

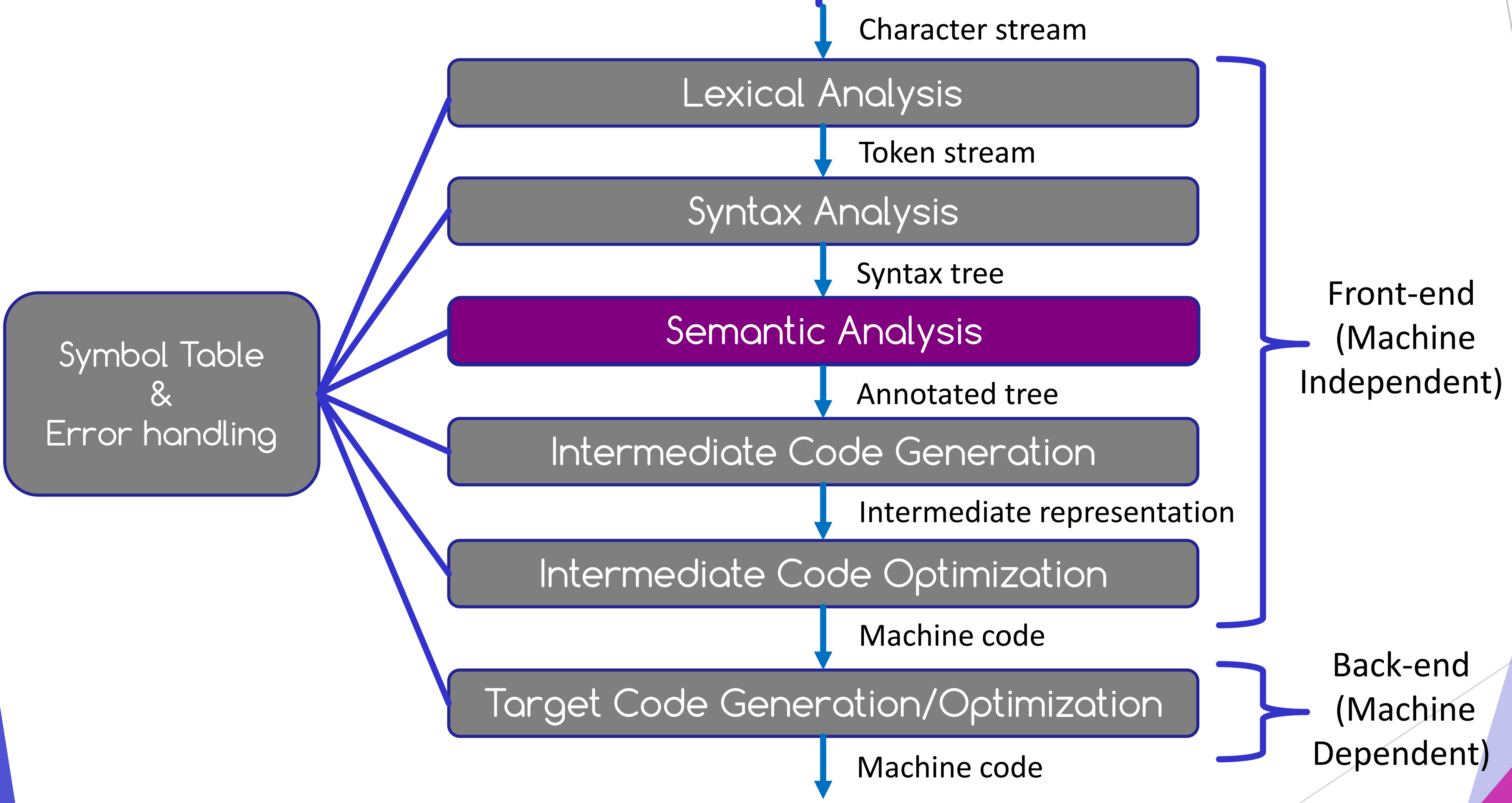
Agenda

- Semantic Analysis
- Attributes and Attribute Grammars
  - Dependency Graphs & Evaluation Order
  - Syntax Directed Definitions
  - Syntax Directed Translation
    - S-attributed SDT
    - L-attributed SDT

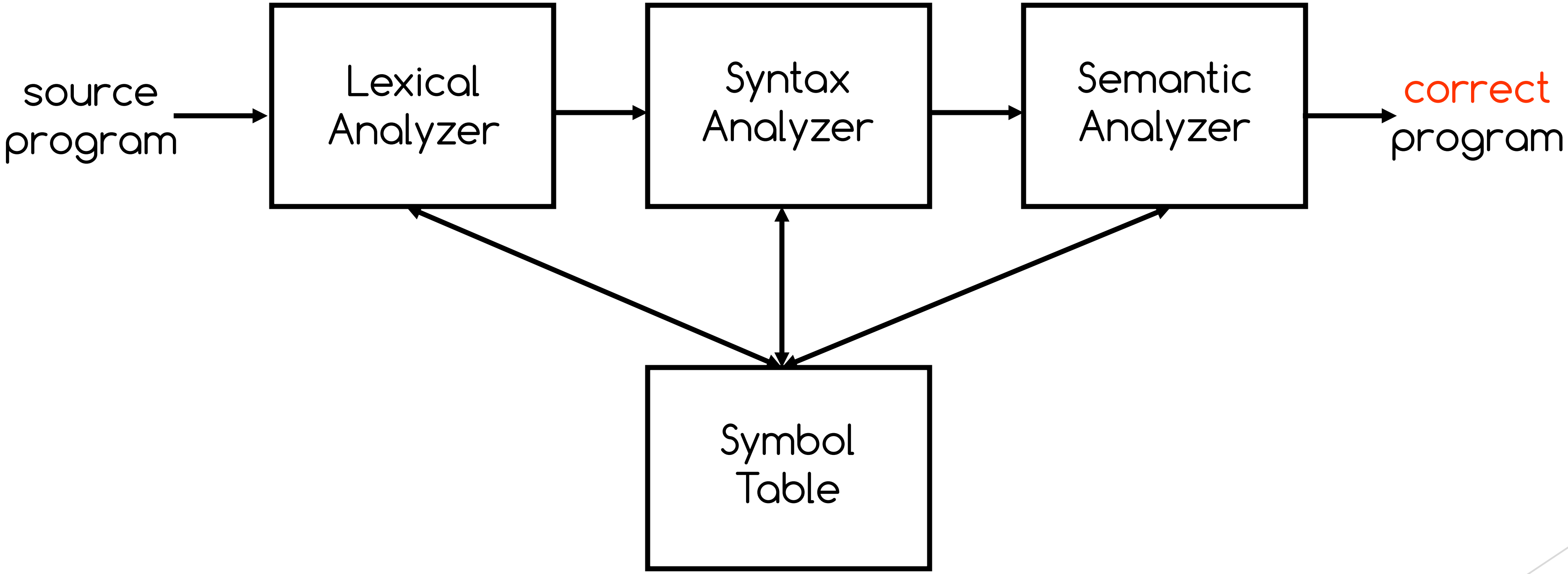
Phases of a Complier



Phases of a Complier



Semantic Analyzer



Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- Context-free grammars used in the syntax analysis are integrated with **attributes (semantic rules)**
  - The result is a syntax-directed translation
  - Attribute grammars
- Ex:

  newval := oldval + 12

  - The type of the identifier *newval* must match with type of the expression *(oldval+12)*

Semantics

- Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret **symbols, their types, and their relations with each other**.
- Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

CFG + semantic rules = Syntax Directed Definitions

Semantic Analysis

- For example:

  `int a = "value";`

- should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs.
- These rules are set by the grammar of the language and evaluated in semantic analysis.
- The following tasks should be performed in semantic analysis:
  - Scope resolution
  - Type checking
  - Array-bound checking

Semantic Errors

Some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Attributes & Attribute Grammar

Attributes

- Any property of a programming language construct such as
  - The data type of a variable
  - The value of an expression
  - The location of a variable in memory
  - The object code of a procedure
  - The number of significant digits in a number
- **Binding of the attribute**
  - The process of **computing an attribute** and **associating its computed value** with the language construct in question

Binding time

- The **time during the compilation/execution process** when the binding of an attribute occurs
- Based on the difference of the binding time, attributes are divided into
  - **Static** attributes (be bound prior to execution)
  - **Dynamic** attributes (be bound during execution)

Example

- The binding time and significance during compilation of the attributes.
  - Attribute computations are extremely varied
  - Type checker
    - In a language like C or Java, is an important part of semantic analysis;
    - While in a language like LISP, data types are dynamic, LISP compiler must generate code to compute types and perform type checking during program execution.
  - The values of expressions
    - Usually dynamic and the be computed during execution
    - But sometime can also be evaluated during compilation (constant folding)

CFG + semantic rules = Syntax Directed Translation

Attribute Grammar

- Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.
- Attribute grammar is a medium to **provide semantics** to the context-free grammar and it can help specify the syntax and semantics of a programming language.
- Attribute grammar (when viewed as a parse-tree) can **pass values or information** among the nodes of a tree.

CFG + semantic rules = Syntax Directed Translation

Adding Attributes to Production Rules of CFG

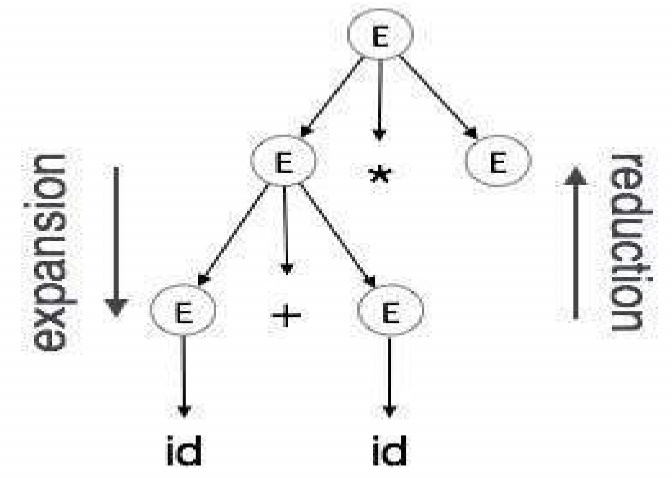
$E \rightarrow E + T$

{ E.value = E.value + T.value }

Production Rule

- The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted.
- Here, the values of non-terminals E and T are added together, and the result is copied to the non-terminal E.

Expansion and Reduction



- Expansion: When a non-terminal is expanded to terminals as per a grammatical rule
- Reduction: When a terminal is reduced to its corresponding non-terminal according to grammar rules.

Reduction

- Syntax trees are parsed **top-down** and **left to right**. Whenever reduction occurs, we apply its corresponding semantic rules (**actions**).
- Semantic analysis uses **Syntax Directed Translations** (SDT) to perform the above tasks.
- Semantic analyzer receives AST (**Abstract Syntax Tree**) from its previous stage (syntax analysis).
- Semantic analyzer **attaches attribute information with AST**, which are called **Attributed AST**.
- Attributes are two tuple value, <attribute name, attribute value>
- Example:

  int value = 5;

  <type, “integer”>

  <presentvalue, “5”>

For every production, we attach a semantic rule.

Example 1

consider the following simple grammar for unsigned numbers:

Number → number digit | digit

Digit → 0|1|2|3|4|5|6|7|8|9

The most significant attribute: numeric value (write as val), and the coresponding attribute grammar is as follows:

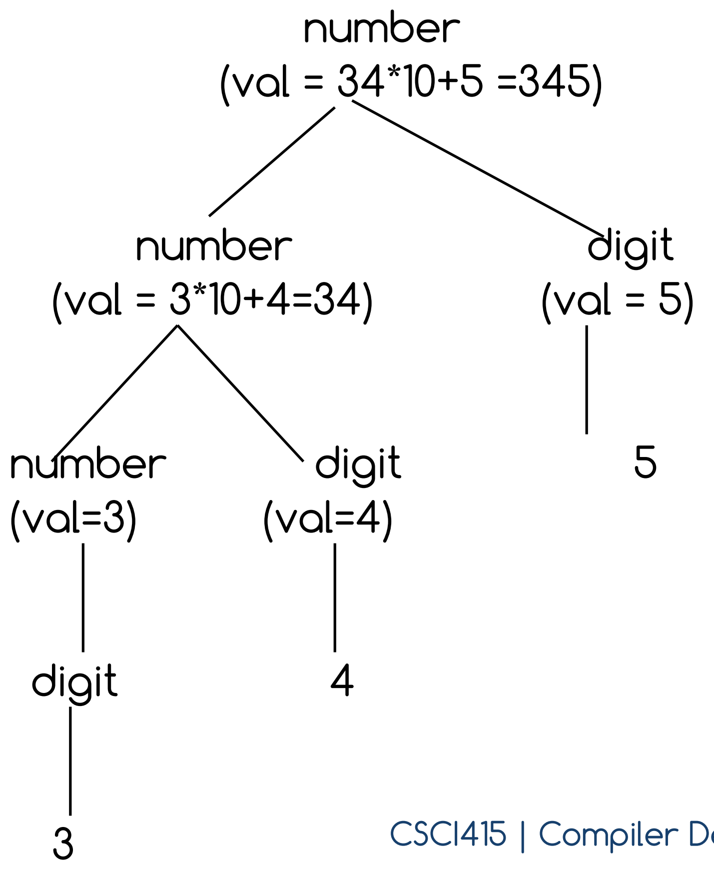
Grammar Rule	Semantic Rules
Number1→number2 digit	number1.val = number2.val*10+digit.val
Number→digit	number.val= digit.val
digit→0	digit.val = 0
digit→1	digit.val = 1
digit→2	digit.val = 2
digit→3	digit.val = 3
digit→4	digit.val = 4
digit→5	digit.val = 5
digit→6	digit.val = 6
digit→7	digit.val = 7
digit→8	digit.val = 8
digit→9	digit.val = 9

The parse tree showing attribute computations for the number 345 is given as follows

Number→Number digit	number1.val = number2.val*10+digit.val
Number→digit	number.val= digit.val

The parse tree showing attribute computations for the number 345 is given as follows

Number→Number digit	number1.val = number2.val*10+digit.val
Number→digit	number.val= digit.val



Example 2

Consider the following simple grammar of variable declarations in a C-like syntax:

Decl → type var-list
Type→ int | float
Var-list→ id, var-list |id

Define a data type attribute for the variables given by the identifiers in a declaration and write equations expressing how the data type attribute is related to the type of the declaration.

Example 2

Consider the following simple grammar of variable declarations in a C-like syntax:

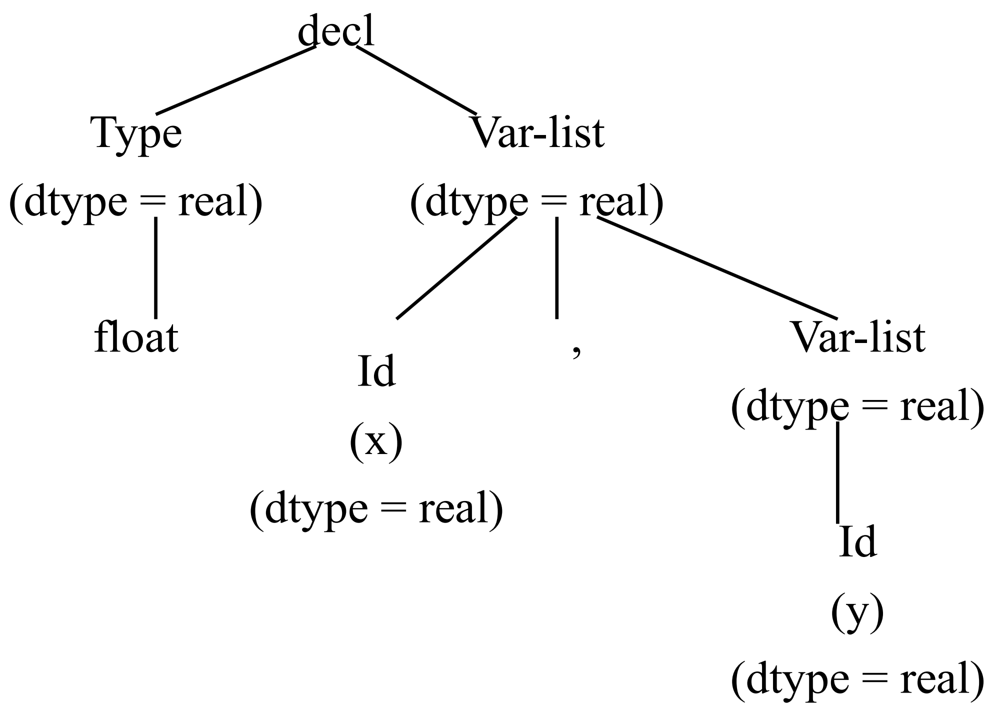
Decl → type var-list
Type→ int | float
Var-list→ id, var-list |id

Define a data type attribute for the variables given by the identifiers in a declaration and write equations expressing how the data type attribute is related to the type of the declaration as follows:

Grammar Rule	Semantic Rules
decl→type var-list	var-list.dtype = type.dtype
type→int	type.dtype = integer
type→ float	type.dtype = real
var-list1→id,var-list2	id.dtype = var-list1.dtype var-list2.dtype= var-list1.dtype
var-list→id	id.type = var-list.dtype

Draw the annotated Parse tree for the string float x,y

Grammar Rule	Semantic Rules
decl→type var-list	var-list.dtype = type.dtype
type→int	type.dtype = integer
type→ float	type.dtype = real
var-list1→id,var-list2	id.dtype = var-list1.dtype
	var-list2.dtype= var-list1.dtype
var-list→id	id.type = var-list.dtype



Dependency Graphs & Evaluation Order

Dependencies of Attributes

- In the semantic rule $b := f(c_1, c_2, \ldots, c_k)$
  we say b *depends on* $c_1, c_2, \ldots, c_k$
- The semantic rule for b must be evaluated *after* the semantic rules for $c_1, c_2, \ldots, c_k$
- The dependencies of attributes can be represented by a directed graph called *dependency graph*

Dependency Graphs

- **Dependency graph** of the string:

  The union of the dependency graphs of the grammar rule choices representing each node (nonleaf) of the parse tree of the string

$$X_i.a_j = f_{ij}(\dots, X_m.a_k, \dots)$$

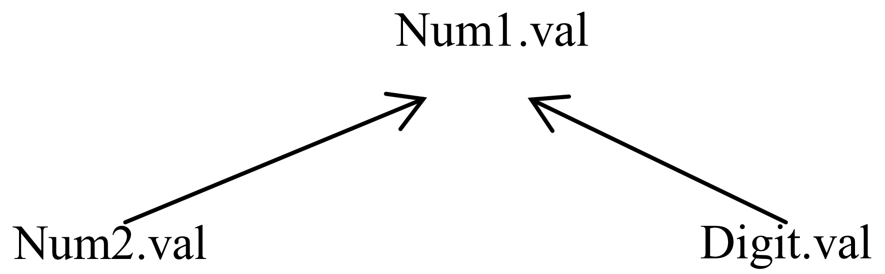
  - An edge from each node $X_m.a_k$ to $X_i.a_j$ the node expressing the dependency of $X_i.a_j$ on $X_m.a_k$.

Dependency Graph of Example 1

Consider the grammar of Example 1, with the attribute grammar below. For each symbol there is only one node in each dependency graph, corresponding to its val attribute

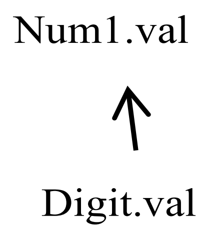
Number→Number digit	number1.val = number2.val*10+digit.val
Number→digit	number.val= digit.val

The dependency graph for this grammar rule choice is

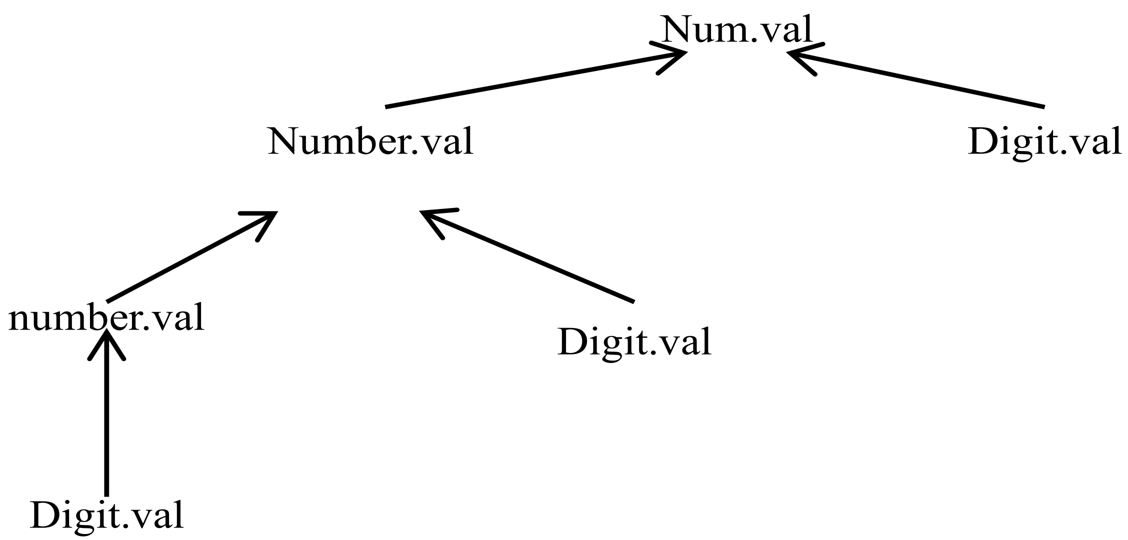


The subscripts for repeated symbols will be omitted

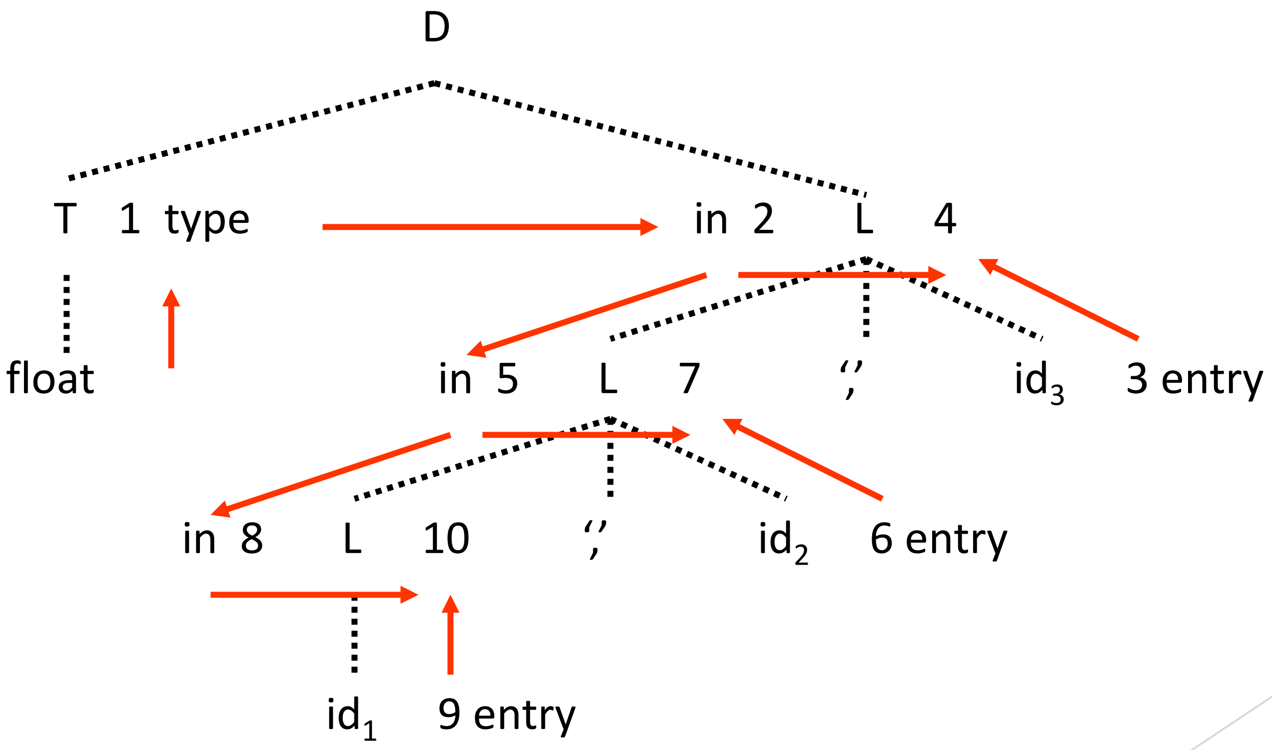
Number→ digit number.val = digit.val



The string 345 has the following dependency graph.



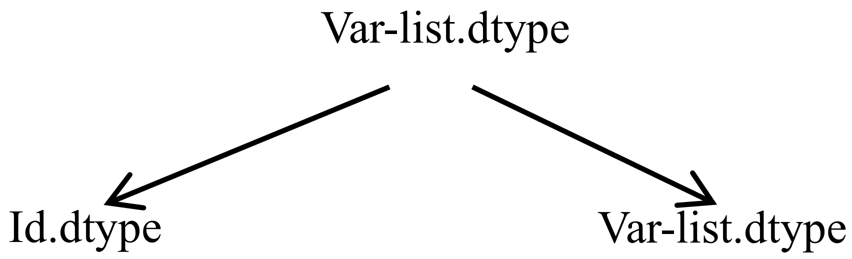
Dependency Graphs



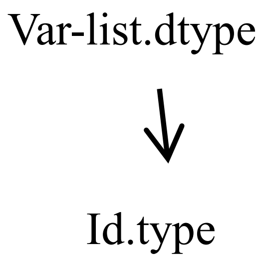
Dependency Graph of Example 2

Grammar Rule	Semantic Rules
decl→type var-list	var-list.dtype = type.dtype
type→int	type.dtype = integer
type→ float	type.dtype = real
var-list1→id,var-list2	id.dtype = var-list1.dtype var-list2.dtype= var-list1.dtype
var-list→id	id.type = var-list.dtype

The dependency graph



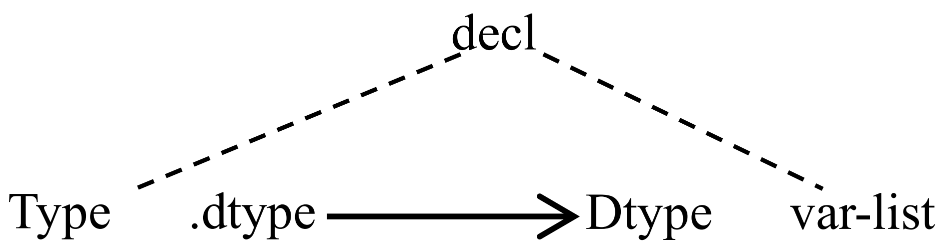
similarly var-list →id respond to



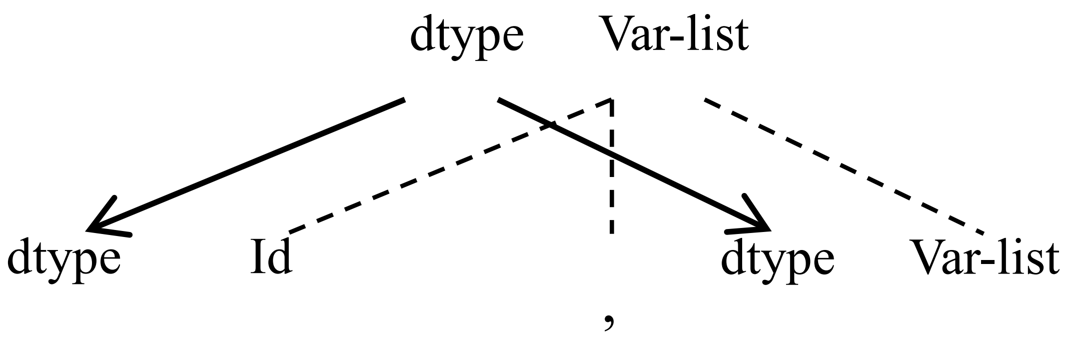
Decl→type varlist

Type.dtype -> var-list.dtype

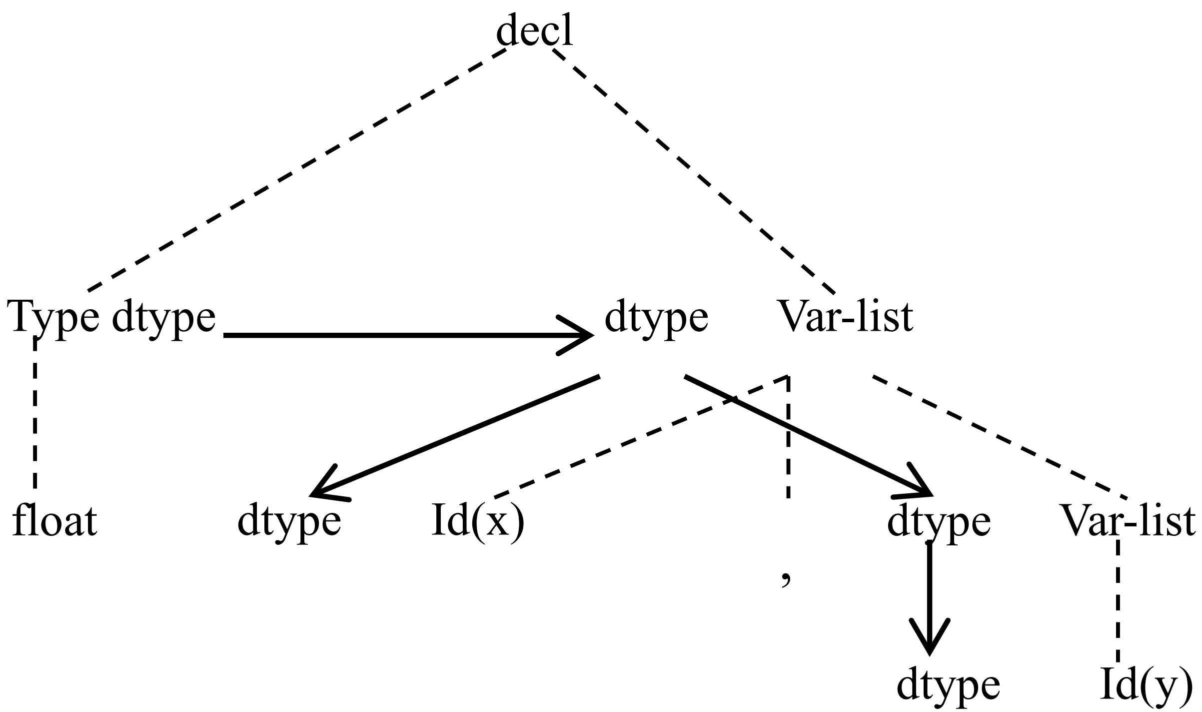
It can also be drawn as:



So, the first graph in this example can be drawn as :



Finally, the dependency graph for the string float x, y is




Q
A

Synthesized and Inherited Attributes

Types of Attributes

- Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions.
- Based on the way the attributes get their values, they can be broadly divided into two categories:
  - Synthesized attributes
  - Inherited attributes

Syntax-Directed Definitions

- Each grammar production $A \to \alpha$ is associated with a set of semantic rules of the form
  $b := f(c_1, c_2, \ldots, c_k)$
  where f is a function and
  1. b is a *synthesized* attribute of A and $c_1, c_2, \ldots, c_k$ are attributes of A or grammar symbols in α, or
  2. b is an *inherited* attribute of one of the grammar symbols in α and $c_1, c_2, \ldots, c_k$ are attributes of A or grammar symbols in α

Synthesized attributes

- These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

$$S \rightarrow ABC$$

- If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.
- As in our previous example ($E \rightarrow E + T$), the parent node E gets its value from its child node.
- Synthesized attributes never take values from their parent nodes or any sibling nodes.

Inherited Attributes

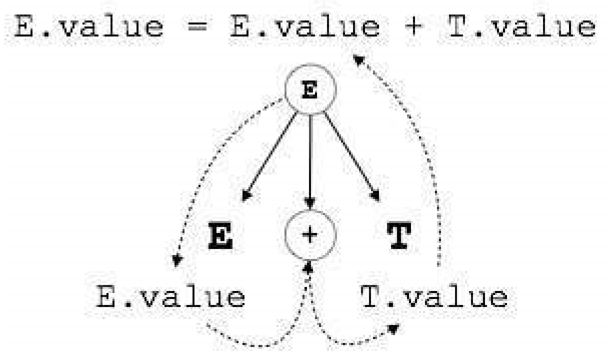
- In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

$$S \rightarrow ABC$$

- A can get values from S, B and C.
- B can take values from S, A, and C.
- C can take values from S, A, and B.

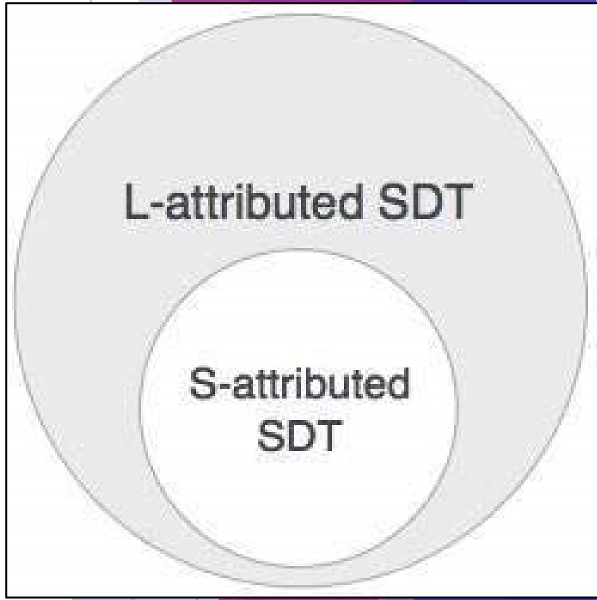
S-attributed Syntax Directed Translation (SDT)

- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
- These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).
- S-attributed SDTs are evaluated in **bottom-up parsing**, as the values of the parent nodes depend upon the values of the child nodes.



L-attributed SDT

- This form of SDT uses **both synthesized and inherited attributes** with restriction of ***not taking values from right siblings***.
  - In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes.
- Attributes in L-attributed SDTs are evaluated by **depth-first and left-to-right** parsing manner.
- We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.



Example

$$S \rightarrow ABC$$

- S can take values from A, B, and C (synthesized).
- A can take values from S only.
- B can take values from S and A.
- C can get values from S, A, and B.
- No non-terminal can get values from the sibling to its right.

S-Attributed SDT Example

Production	Semantic Rules
L → E '\n'	print(E.val)
E → E_1 '+' T	E.val := E_1.val + T.val
E → T	E.val := T.val
T → T_1 '*' F	T.val := T_1.val * F.val
T → F	T.val := F.val
F → '(' E ')'	F.val := E.val
F → digit	F.val := digit.val

The attribute val represents the value of an expression

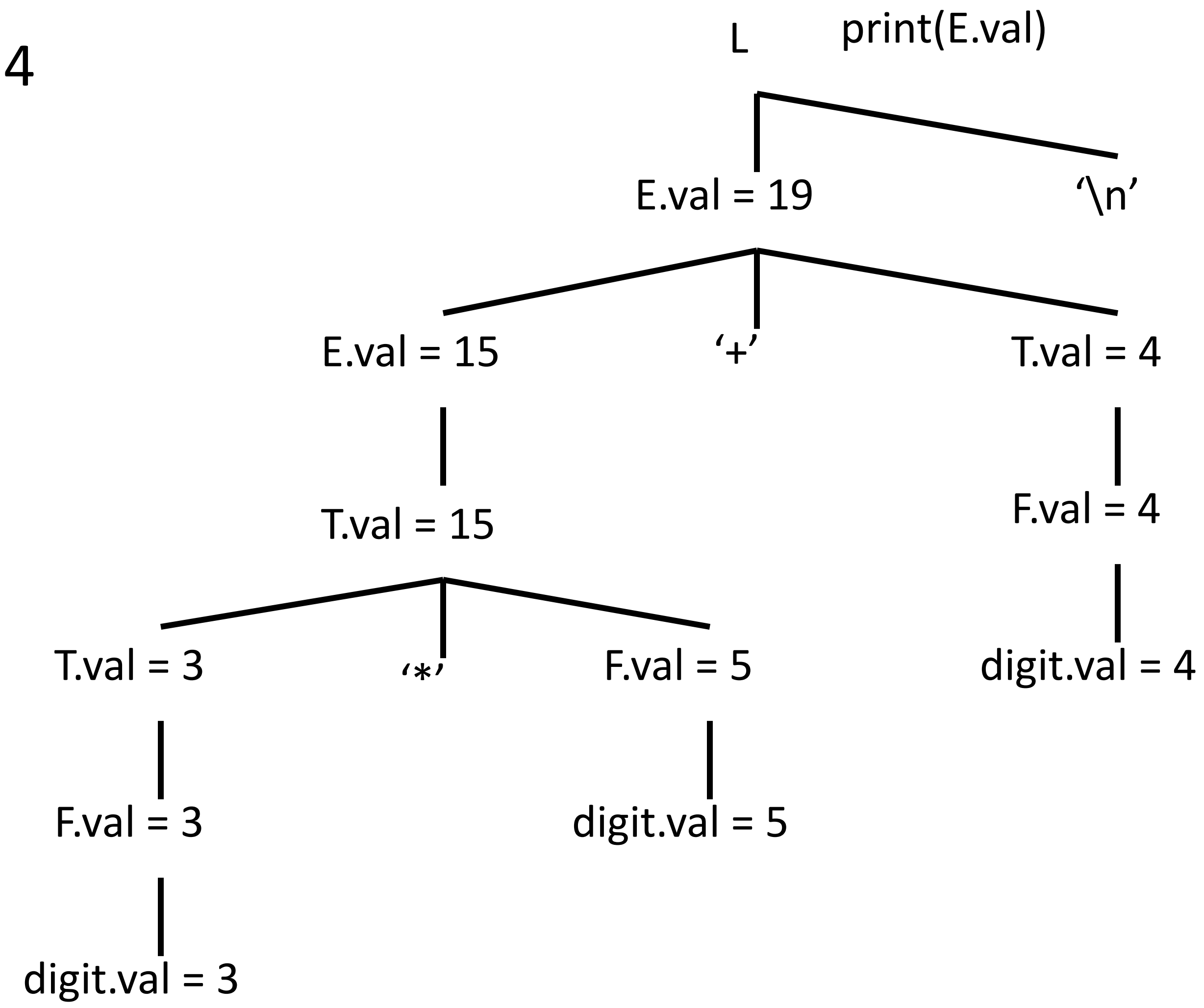
Annotated Parse Trees

3 * 5 + 4

Production	Semantic Rules
L → E '\n'	print(E.val)
E → E_1 '+' T	E.val := E_1.val + T.val
E → T	E.val := T.val
T → T_1 '*' F	T.val := T_1.val * F.val
T → F	T.val := F.val
F → '(' E ')'	F.val := E.val
F → digit	F.val := digit.val

Annotated Parse Trees

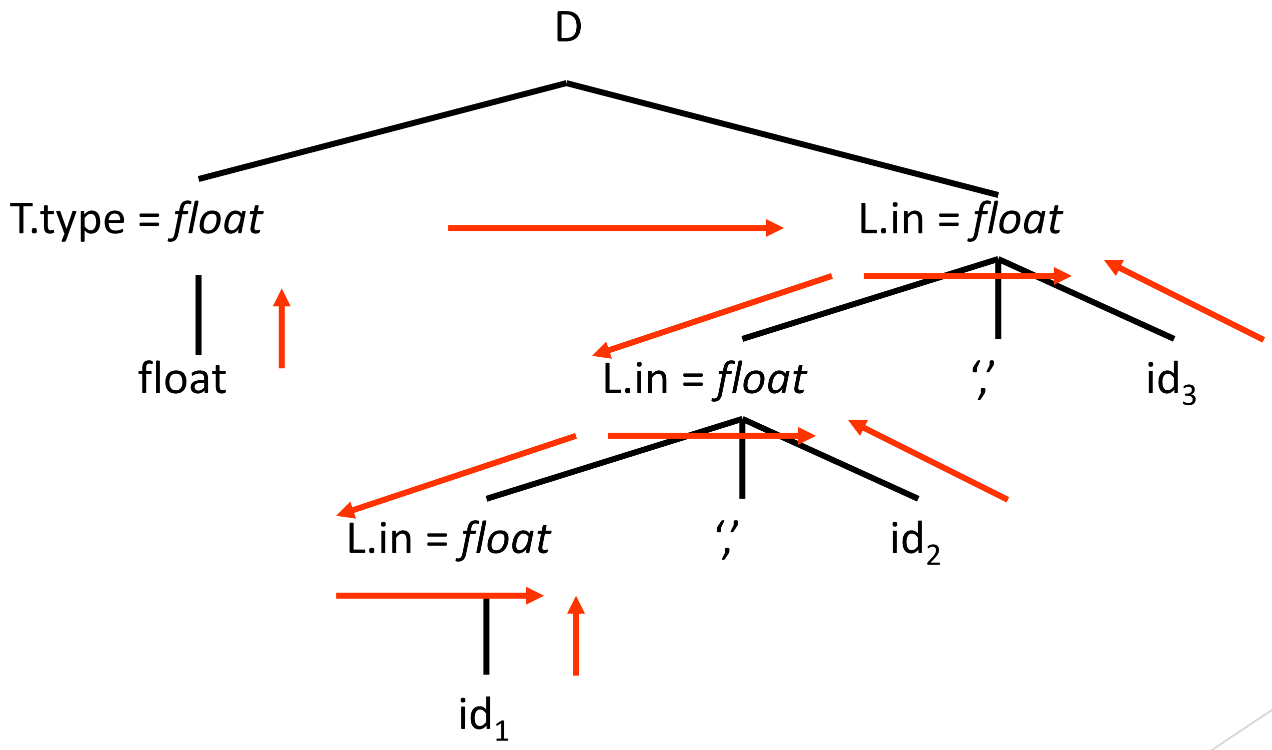
3 * 5 + 4



L-attributed SDT Example

Production	Semantic Rules
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := integer
$T \rightarrow$ **float**	**T.type := float**
$L \rightarrow L_1$ ‘,’ **id**	L_1.in := L.in addtype(**id**.entry, L.in)
$L \rightarrow$ **id**	addtype(**id**.entry, L.in)

Inherited Attributes



Summary: Semantic Analysis

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- Gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
- An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

Summary

- An attribute of a node (grammar symbol) in the parse tree is *synthesized* if its value is computed from that of its children
- An attribute of a node in the parse tree is *inherited* if its value is computed from that of its parent and siblings

S-attributed SDT	L-attributed SDT
It uses only synthesized attributes.	It uses both synthesized and inherited attributes. Each inherited attribute is restricted to inherit from parent or left siblings only
Semantics actions are placed at right end of production. A → BCC { }	Semantics actions are placed at anywhere on RHS. A → { } BC | D { } E | FG { }
S-attributes can be evaluated during bottom-up parsing.	L-attributes are evaluated by traversing the parse tree in depth first, left to right.

Review Questions

Question 1

Consider the given below SDT.

P1: S -> MN {S.val= M.val + N.val}

P2: M -> PQ {M.val = P.val * Q.val and P.val =Q.val}

Select the correct option.

A. Both P1 and P2 are S attributed.
B. P1 is S attributed and P2 is L-attributed.
C. P1 is L attributed but P2 is not L-attributed.

Question 2

- The following grammar generates binary numbers

$N \rightarrow L$

$L \rightarrow LB \mid B$

$B \rightarrow 0 \mid 1$

Design an L-attributed SDT to compute N.val; the decimal number value of the input string.

For example the translation of string 1011 should be 11.

Solution

$N \rightarrow L$

$L \rightarrow LB$

$L \rightarrow B$

$B \rightarrow 0$

$B \rightarrow 1$

Solution

$N \to L \quad \{N.val = L.val\}$

$L \to LB \quad \{L.val = L1.val*2 + B.val\}$

$L \to B \quad \{L.val = B.val\}$

$B \to 0 \quad \{B.val = 0\}$

$B \to 1 \quad \{B.val = 1\}$

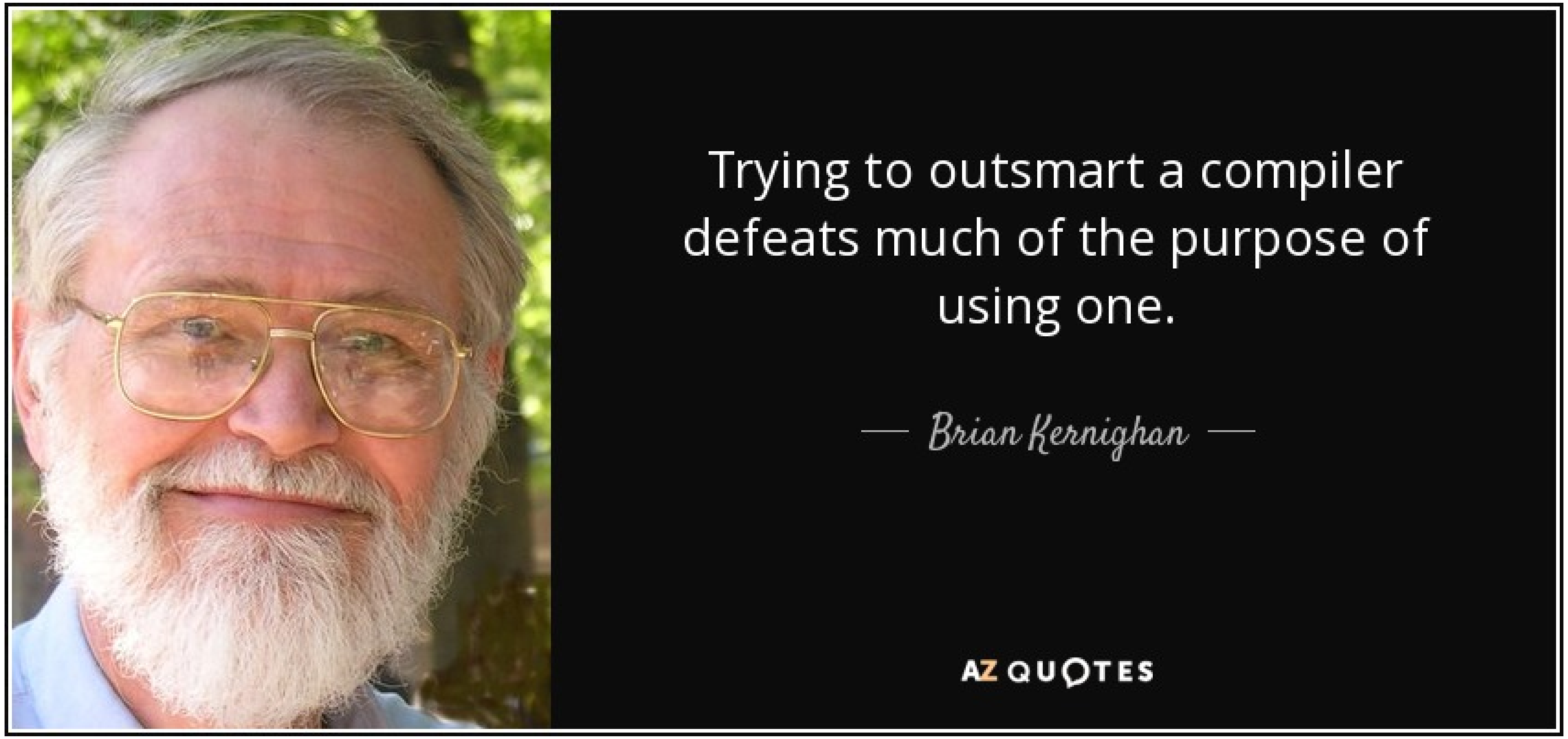
The SDT of the binary number 1011

$N \rightarrow L$	{N.val = L.val}
$L \rightarrow LB$	{L.val = L1.val*2 + B.val}
$L \rightarrow B$	{L.val = B.val}
$B \rightarrow 0$	{B.val = 0}
$B \rightarrow 1$	{B.val = 1}


Q
A

Useful Links

- [Syntax directed translation]
- [Compiler Design Lecture 18 -- Examples of SDT]



See you next lecture

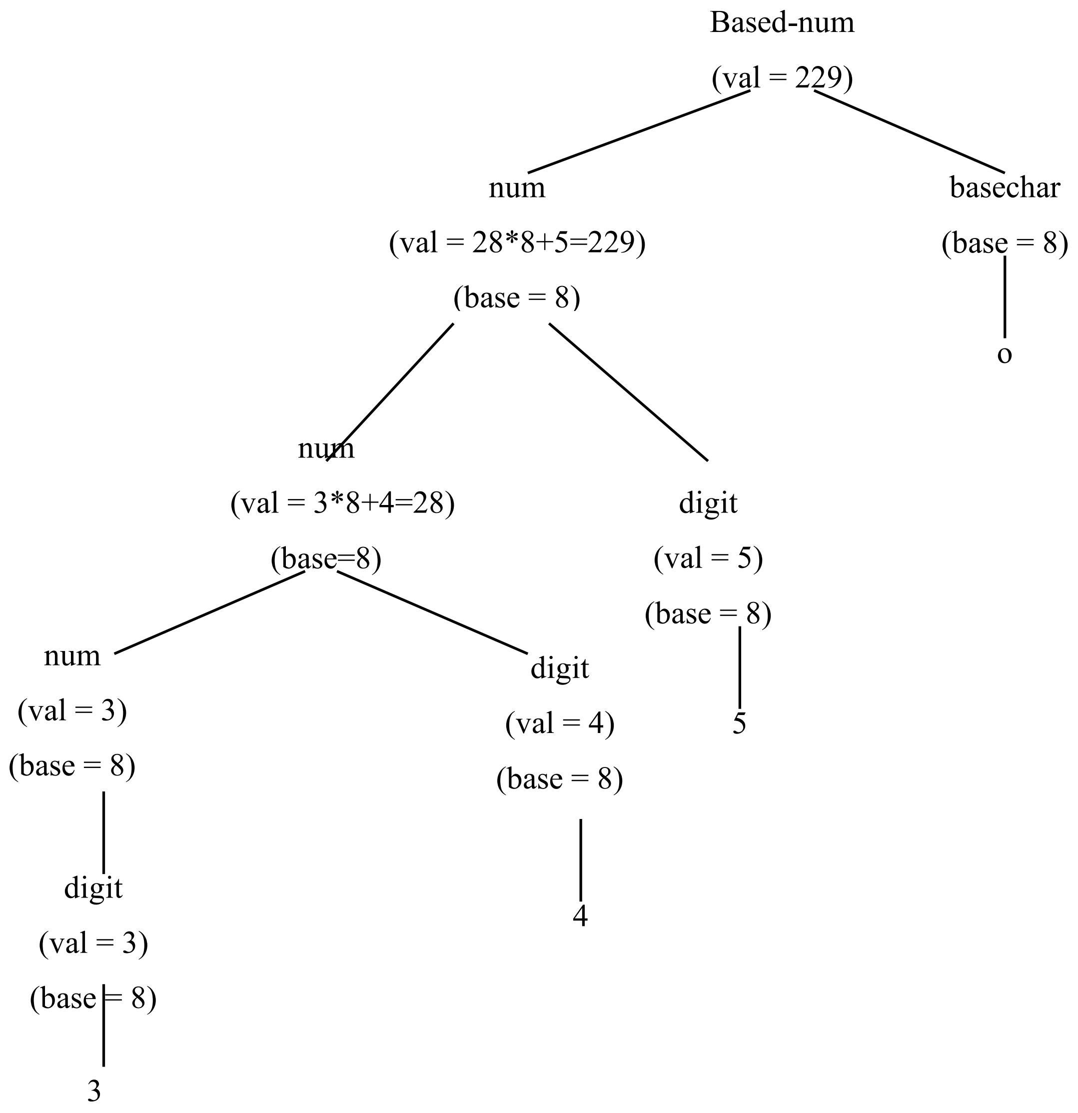
More Examples

Consider the following grammar, where numbers may be octal or decimal, suppose this is indicated by a one-character suffix o(for octal) or d(for decimal):

Based-num → num basechar
Basechar → o|d
Num → num digit | digit
Digit → 0|1|2|3|4|5|6|7|8|9

In this case num and digit require a new attribute base, which is used to compute the val attribute. The attribute grammar for base and val is given as follows.

Grammar Rule	Semantic Rules
Based-num→num basechar	Based-num.val = num.val Based-num.base = basechar.base
Basechar →o	Basechar.base = 8
Basechar→ d	Basechar.base = 10
Num1→num2 digit	num1.val = If digit.val = error or num2.val = error Then error Else num2.val*num1.base+digit.val Num2.base = num1.base Digit.base = num1.base
Num → digit	num.val = digit.val Digit.base = num.base
Digit →0	digit.val = 0
Digit →1	digit.val = 1
…	…
Digit →7	digit.val = 7
Digit →8	digit.val = if digit.base = 8 then error else 8
Digit →9	digit.val = if digit.base = 8 then error else 9


Based-num
(val = 229)
num
(val = 28*8+5=229)
(base = 8)
basechar
(base = 8)
o
num
(val = 3*8+4=28)
(base=8)
digit
(val = 5)
(base = 8)
5
num
(val = 3)
(base = 8)
digit
(val = 4)
(base = 8)
4
digit
(val = 3)
(base = 8)
3