Compilation Principle
编 译 原 理

第14讲：语义分析(2)

张献伟

xianweiz.github.io

DCS290, 4/12/2022

Review Questions

- Why context analysis is not performed in parsing stage?

  Parsing relies on CFG, which is context free.

- Give some examples of semantic analysis.

  Def-before-use, no redefinition, same type, scoping …

- What is Syntax Directed Translation?

  The parsing process and parse trees are to direct semantic analysis and the translation of the program (a.k.a., CFG-driven translation)

- How to augment grammar for semantic analysis?

  Semantic attributes for symbols, rules/actions for productions

- What are SDD and SDT?

  SDD = Syntax Directed Definitions, SDT = SD Translation Schemes

- What is an synthesized attribute?

  Defined by attribute values of node *N*'s children and *N* itself

Example: Synthesized Attribute (cont.)

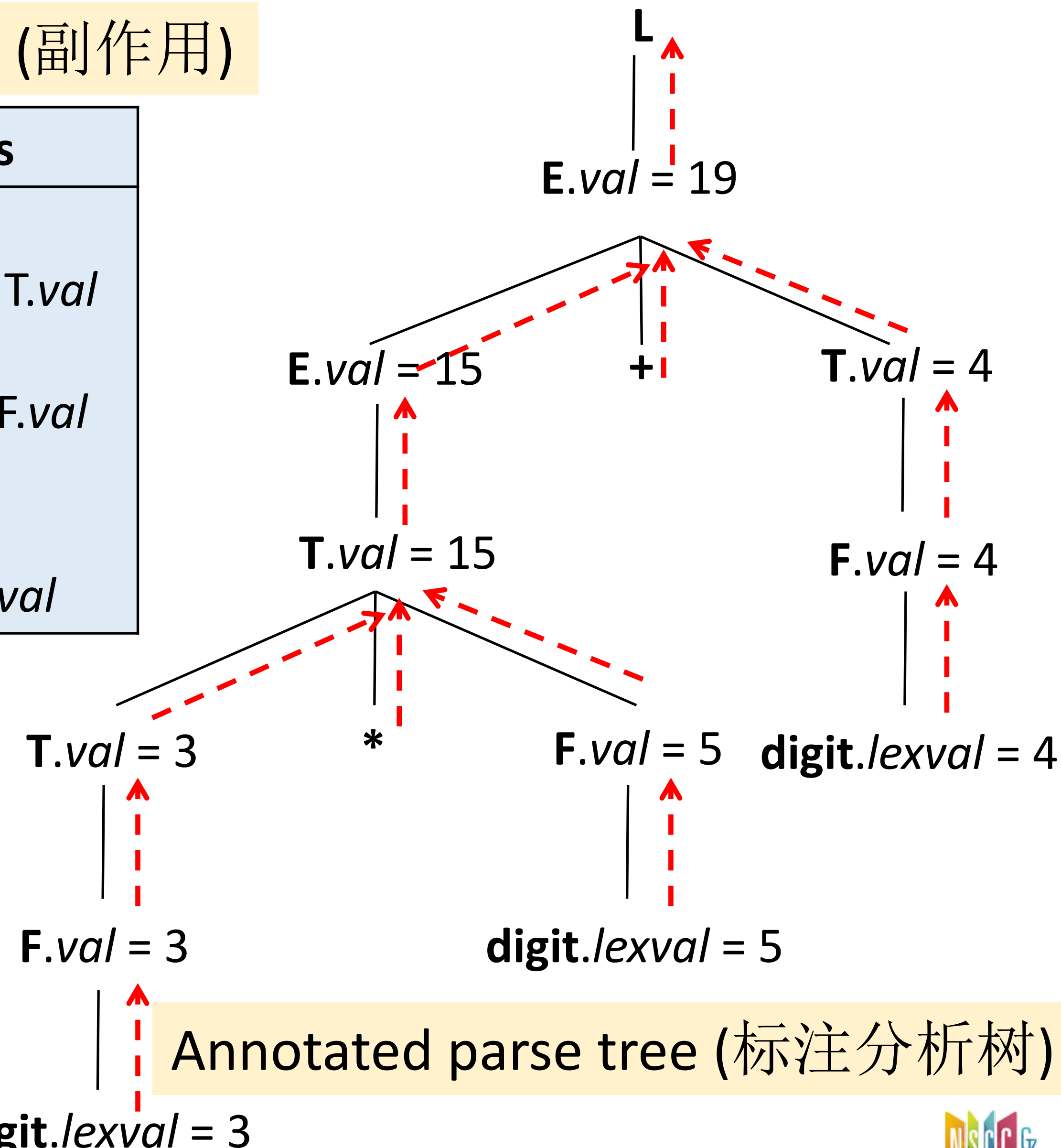
SDD:

Side effect (副作用)

Production Rules	Semantic Rules
(1) L -> E	print(E.*val*)
(2) E -> E_1 + T	E.*val* = E_1.*val* + T.*val*
(3) E -> T	E.*val* = T.*val*
(4) T -> T_1 * F	T.*val* = T_1.*val* x F.*val*
(5) T -> F	T.*val* = F.*val*
(6) F -> (E)	F.*val* = E.*val*
(7) F -> digit	F.*val* = digit.*lexval*

Input:

3 * 5 + 4



Annotated parse tree (标注分析树)

Example: Inherited Attribute[继承]

SDD:

Production Rules	Semantic Rules
(1) D -> T L	L.*inh* = T.*type*
(2) T -> int	T.*type* = int
(3) T -> float	T.*type* = float
(4) L -> L_1, id	L_1.*inh* = L.*inh* addtype(id.*entry*, L.*inh*)
(5) L -> id	addtype(id.*entry*, L.*inh*)

T has synthesized attribute *type*
L has inherited attribute *inh*

Pointing to a symbol-table[符号表] object

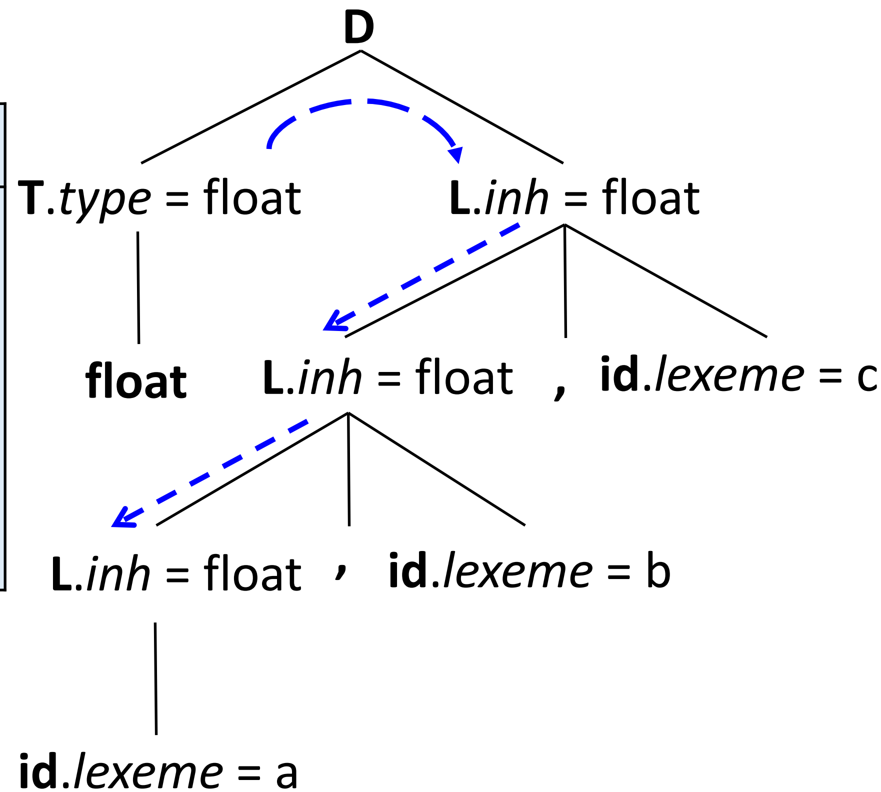
Variable declaration of type int/float followed by a list of IDs:

(1) Declaration: a type *T* followed by a list of *L* identifiers
(2) Evaluate the synthesized attribute *T.type* (int)
(3) Evaluate the synthesized attribute *T.type* (float)
(4) Pass down type, and add type to symbol table entry for the identifier
(5) Add type to symbol table

Example: Inherited Attribute (cont.)

SDD:

Production Rules	Semantic Rules
(1) D -> T L	L.*inh* = T.*type*
(2) T -> int	T.*type* = int
(3) T -> float	T.*type* = float
(4) L -> L_1, id	L_1.*inh* = L.*inh* addtype(id.*entry*, L.*inh*)
(5) L -> id	addtype(id.*entry*, L.*inh*)



Input:

float a, b, c

type depends on child
inh depends on sibling or parent

The Concepts

- **Side effect**[副作用]
  - 一般属性值计算（基于属性值或常量进行的）之外的功能
  - 例如：code generation, print results, modify symbol table …
- **Attribute grammar**[属性文法]
  - 一个没有副作用的SDD
  - The rules define the value of an attribute purely in terms of the value of other attributes and constants[属性文法的规则仅仅通过其他属性值和常量来定义一个属性值]
- **Annotated parse-tree**[标注分析树]
  - 每个节点都带有属性值的分析树
    - A parse tree showing the value(s) of its attribute(s)
  - a.k.a., attribute parse tree[属性分析树]
  - Can also have actions being annotated[也可标注语义动作]

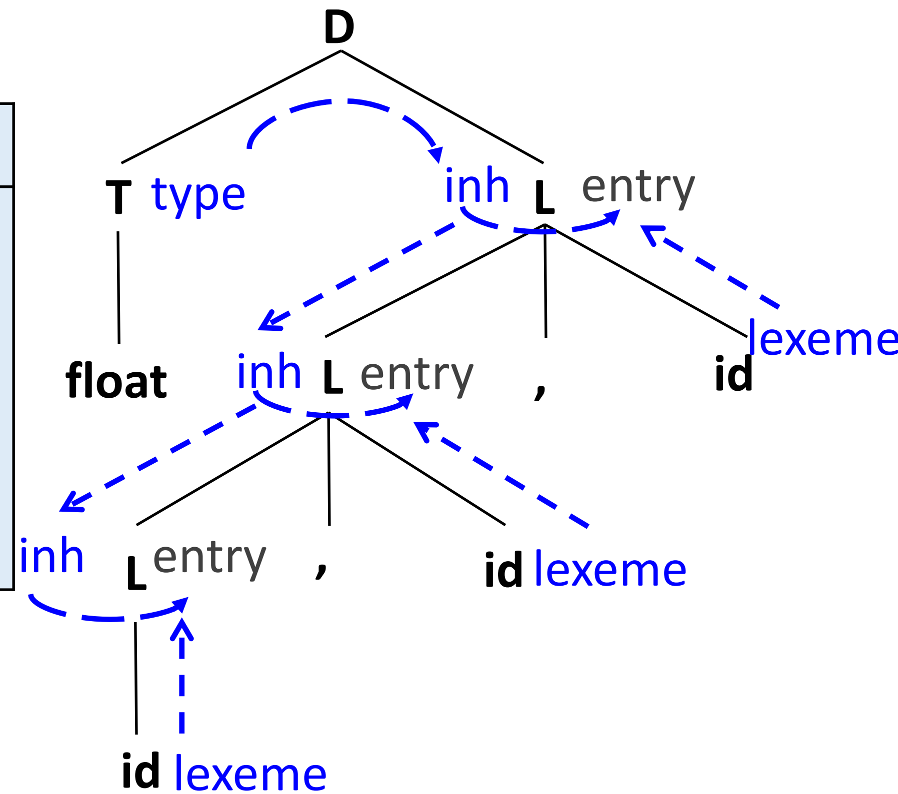
Dependence Graph[依赖图]

- Dependence relationship[依赖关系]
  - Before evaluating an attribute at a node of a parse tree, we must evaluate all attributes it depends on[按照依赖顺序计算]
- **Dependency graph**[依赖图]
  - While the annotated parse tree shows the values of attributes, a dependency graph helps determine how those values can be computed[依赖图决定属性值的计算]
  - Depicts the flow of info among the attribute instances in a particular parse tree[描绘了分析树的属性信息流]
    - **Directed graph** where edges are dependence relationships between attributes
    - For each parse-tree node *X*, there's a graph node for each attr of *X*
    - If attr *X.a* depends on attr *Y.b*, then there's one directed edge from *Y.b* to *X.a*

Example: Dependency Graph

SDD:

Production Rules	Semantic Rules
(1) D -> T L	L.*inh* = T.*type*
(2) T -> int	T.*type* = int
(3) T -> float	T.*type* = float
(4) L -> L_1, id	L_1.*inh* = L.*inh* addtype(id.*entry*, L.*inh*)
(5) L -> id	addtype(id.*entry*, L.*inh*)



Input:

float a, b, c

‘entry’ is dummy attribute for the *addtype()*

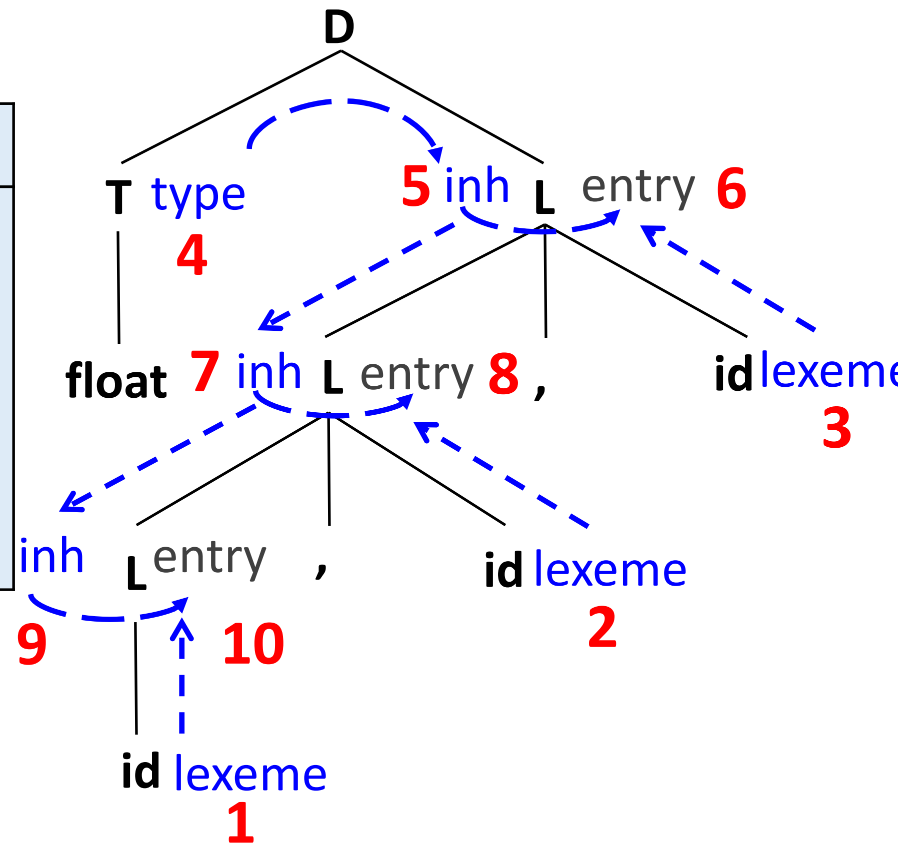
Evaluation Order[属性值计算顺序]

- Ordering the evaluation of attributes[计算顺序]
  - Dependency graph characterizes possible orders in which we can evaluate the attributes at the various nodes of a parse-tree
- If the graph has an edge from node M to node N, then the attribute associated with M must be evaluated before N[用图的边来确定计算顺序]
  - Thus, the only allowable orders of evaluation are those sequences of nodes N_1, N_2, …, N_k such that if there is an edge of the graph from N_i to N_j, then i < j
  - Such an ordering embeds a directed graph into a linear order, and is called a **topological sort**[拓扑排序] of the graph
    - If there's any cycle in the graph, then there are no topological sorts, i.e., no way to evaluate the SDD on this parse tree
    - If there are no cycles, then there is always at least one topological sort

Example: Evaluation Order

SDD:

Production Rules	Semantic Rules
(1) D -> T L	L.*inh* = T.*type*
(2) T -> int	T.*type* = int
(3) T -> float	T.*type* = float
(4) L -> L_1, id	L_1.*inh* = L.*inh* addtype(id.*lexeme*, L.*inh*)
(5) L -> id	addtype(id.*lexeme*, L.*inh*)



Input:

float a, b, c

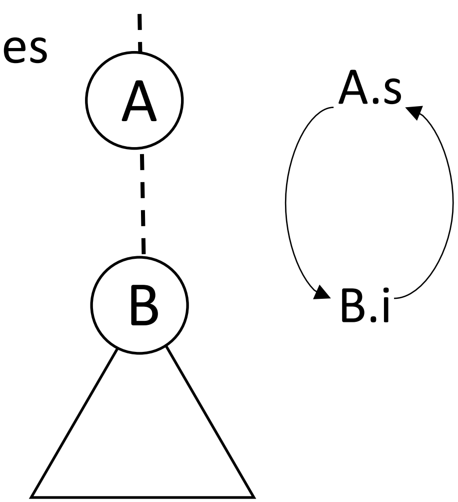
Topological sort:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Evaluation Order (cont.)

- Before evaluating an attribute at a node of a parse tree, we must evaluate all attributes it depends on[依赖关系]
  - Synthesized: evaluate children first, then the node itself
    - Any bottom-up order is fine
  - For SDD's with both inherited and synthesized attributes, there's no guarantee that there is even one evaluation order
- Difficult to determine whether exist any circularities[非常难确定是否有循环依赖]
  - But, there are useful subclasses of SDD's that are sufficient to guarantee that an evaluation order exists
    - Such classes do not permit graphs with cycles

Production	Semantic Rules
A -> B	A.*s* = B.*i*;
	B.*i* = A.*s* + 1;



S-Attributed Definitions[S-属性定义]

- An SDD is **S-attributed** if every attribute is synthesized[只具有综合属性]
- If an SDD is S-attributed (S-SDD)
  - We can evaluate its attributes in any bottom-up order of the nodes of the parse-tree[任何自底向上的顺序计算属性值]
  - Can be implemented during bottom-up parsing[LR分析中实现]

Production Rules	**Semantic Rules**
(1) L -> E	print(E.*val*)
(2) E -> E_1 + T	E.*val* = E_1.*val* + T.*val*
(3) E -> T	E.*val* = T.*val*
(4) T -> T_1 * F	T.*val* = T_1.*val* x F.*val*
(5) T -> F	T.*val* = F.*val*
(6) F -> (E)	F.*val* = E.*val*
(7) F -> digit	F.*val* = digit.*lexval*

L-Attributed Definitions[L-属性定义]

- An SDD is **L-attributed** (L-SDD) if
  - Between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left[依赖图的边只能从左到右]
  - More precisely: each attribute must be either **synthesized**, or **inherited** but with the rules limited as follows: suppose A -> $X_1X_2 \ldots X_n$, the inherited attribute $X_i.a$ only depends on
    - **Inherited** attributes associated with A
    - Either *syn* or *inh* attributes of $X_1, X_2, \ldots, X_{i-1}$ located to the left of X_i
    - Either *syn* or *inh* attributes of X_i itself, but no cycles formed by the attributes of this X_i

Why not synthesized?

Cycle: X_i depends on A, A.*s* depends on X_i

- Can be implemented during top-down parsing[LL分析中]

S-SDD or L-SDD?

Production Rules	Semantic Rules
A -> B C	A.*s* = B.*b* B.*i* = f(C.*c*, A.*s*)

Not L-SDD: A.*s* is syn attr

Not S-SDD: B.*i* is inh

Not L-SDD: C is right to B

Syntax Directed Trans. Impl.[实现]

- Learnt how to specify translation: SDD and SDT[定义]
  - SDT is an executable specification of the SDD
    - CFG with semantic actions embedded in production bodies

- SDT can be implemented in two ways[具体实现]
  - Using a parse tree or AST[基于预先构建的分析树]
    - First build a parse tree, and then apply rules or actions at each node while traversing the tree
    - All SDDs (without cycles) and SDTs can be implemented
      - Since the tree can be traversed freely, implements any ordering
  - During parsing, without building a parse tree[语法分析过程中]
    - Apply rules or actions at each production while parsing
    - Only a subset of SDDs and SDTS can be implemented
      - Evaluation ordering restricted to parser derivation order

Syntax Directed Trans. Impl. (cont.)

- Typically, SDD (i.e., semantic analysis) is implemented during parsing[更为高效]
  - Allows compiler to skip parse tree generation
  - Saves time and memory

- Two important classes of SDD's[两个关键子类]
  - SDD is *S-attributed*, the underlying grammar is *LR-parsable*
  - SDD is *L-attributed*, the underlying grammar is *LL-parsable*
  - For both classes, semantic rules in an SDD can be converted into an SDT with actions that are executed at the right time[允许SDD到SDT的转换]
    - During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of the action have been matched

== Implement S-SDD ==

- Convert S-attributed SDD to SDT by[SDD->SDT的转换]
  - Placing each action at the end of the production[将每个语义动作都放在产生式的最后]
  - SDTs with all actions at the right ends of the production bodies are called **postfix SDT's**[后缀/尾部SDT]

S-SDD

Production Rules	Semantic Rules
(1) L -> E	print (E.*val*)
(2) E -> E_1 + T	E.*val* = E_1.*val* + T.*val*
(3) E -> T	E.*val* = T.*val*
(4) T -> T_1 * F	T.*val* = T_1.*val* x F.*val*
(5) T -> F	T.*val* = F.*val*
(6) F -> (E)	F.*val* = E.*val*
(7) F -> digit	F.*val* = digit.*lexval*

SDT

CFG with actions
(1) L -> E { print (E.*val*) }
(2) E -> E_1 + T { E.*val* = E_1.*val* + T.*val* }
(3) E -> T { E.*val* = T.*val* }
(4) T -> T_1 * F { T.*val* = T_1.*val* x F.*val* }
(5) T -> F { T.*val* = F.*val* }
(6) F -> (E) { F.*val* = E.*val* }
(7) F -> digit { F.*val* = digit.*lexval* }

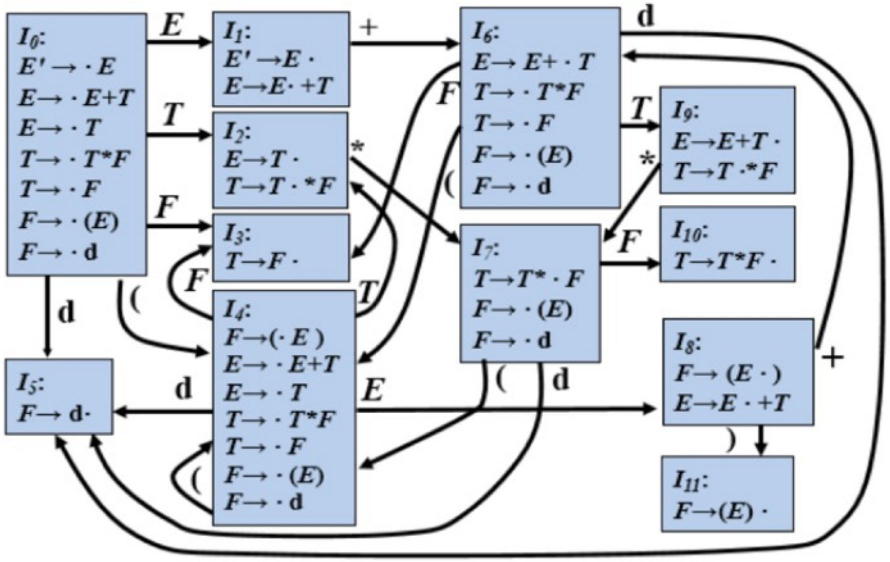
Implement S-SDD (cont.)

- If the underlying grammar of S-SDD is LR parsable
  - Then the SDT can be implemented during LR parsing
- Implement the converted SDT by[借助归约实现]
  - Executing the action along with the reduction of *head <- body*

SDT

CFG with actions
(1) L -> E { print (E.*val*) }
(2) E -> E_1 + T { E.*val* = E_1.*val* + T.*val* }
(3) E -> T { E.*val* = T.*val* }
(4) T -> T_1 * F { T.*val* = T_1.*val* x F.*val* }
(5) T -> F { T.*val* = F.*val* }
(6) F -> (E) { F.*val* = E.*val* }
(7) F -> digit { F.*val* = digit.*lexval* }

SLR Automaton



Extend LR Parse Stack[扩展分析栈]

- Save synthesized attributes into the stack[栈中额外存放综合属性值]
  - Place the attributes along with the grammar symbols (or LR states that associated with these symbols) in records on stack
  - If there are multiple attributes
    - Make the records large enough or by putting pointers to records on the stack[栈记录足够大，或栈记录中存放指针]
- Example: A -> XYZ {action}
  - x, y, z are attributes of X, Y, Z respectively
  - After the action, A and its attributes are at the top (i.e., m-2)

state →	S_0	…	S_{m-2}	S_{m-1}	S_m
symbol →	$	…	X	Y	Z
attribute →	-	…	X.*x*	Y.*y*	Z.*z*
					↑ top

⇒

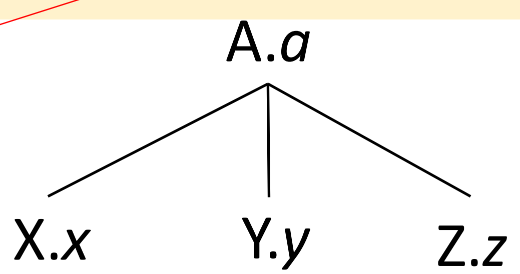
state →	S_0	…	S_n
symbol →	$	…	A
attribute →	-	…	A.*a*
			↑ top

Stack Manipulation[栈操作]

- Rewrite the actions to manipulate the parser stack[语义动作]
  - The manipulation can be done automatically by the parser

stack[top-2].*symbol* = A
stack[top-2].*val* = f(stack[top-2].*val*, stack[top-1].*val*, stack[top].*val*)
top = top -2

A -> XYZ { A.a = f(X.*x*, Y.*y*, Z.*z*) }



state →	S_0	…	S_{m-2}	S_{m-1}	S_m
symbol →	$	…	X	Y	Z
attribute →	-	…	X.*x*	Y.*y*	Z.*z*
					↑ top

state →	S_0	…	S_n
symbol →	$	…	A
attribute →	-	…	A.*a*
			↑ top

中山大學 SUN YAT-SEN UNIVERSITY

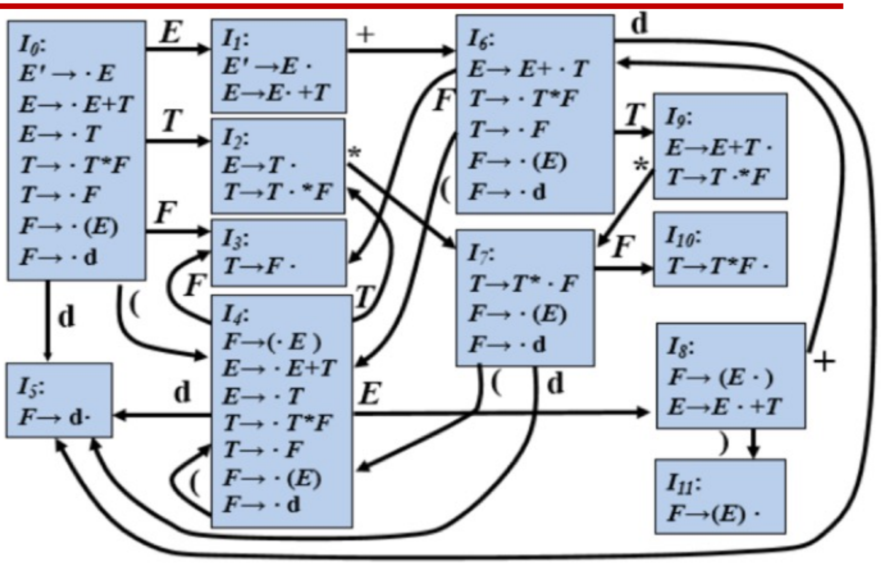
Example

- Rewrite the actions to manipulate the parser stack
  - The manipulation can be done automatically by the parser

Productions	**Semantic Rules**	**Semantic Actions**
(1) L -> E	print (E.*val*)	{ print(stack[top].val); }
(2) E -> E_1+T	E.*val* = E_1.*val*+T.*val*	{ stack[top-2].*val* = stack[top-2].*val* + stack[top].*val*; top = top -2; }
(3) E -> T	E.*val* = T.*val*	
(4) T -> T_1*F	T.*val* = T_1.*val*xF.*val*	{ stack[top-2].*val* = stack[top-2].*val* x stack[top].*val*; top = top -2; }
(5) T -> F	T.*val* = F.*val*	
(6) F -> (E)	F.*val* = E.*val*	{ stack[top-2].*val* = stack[top-1].*val*; top = top -2; }
(7) F -> digit	F.*val* = digit.*lexval*	

Example

Productions	Semantic Actions
(1) L -> E	{ print(stack[top].val); }
(2) E -> E_1+T	{ stack[top-2].*val* = stack[top-2].*val* + stack[top].*val*; top = top -2; }
(3) E -> T	
(4) T -> T_1*F	{ stack[top-2].*val* = stack[top-2].*val* x stack[top].*val*; top = top -2; }
(5) T -> F	
(6) F -> (E)	{ stack[top-2].*val* = stack[top-1].*val*; top = top -2; }
(7) F -> digit	



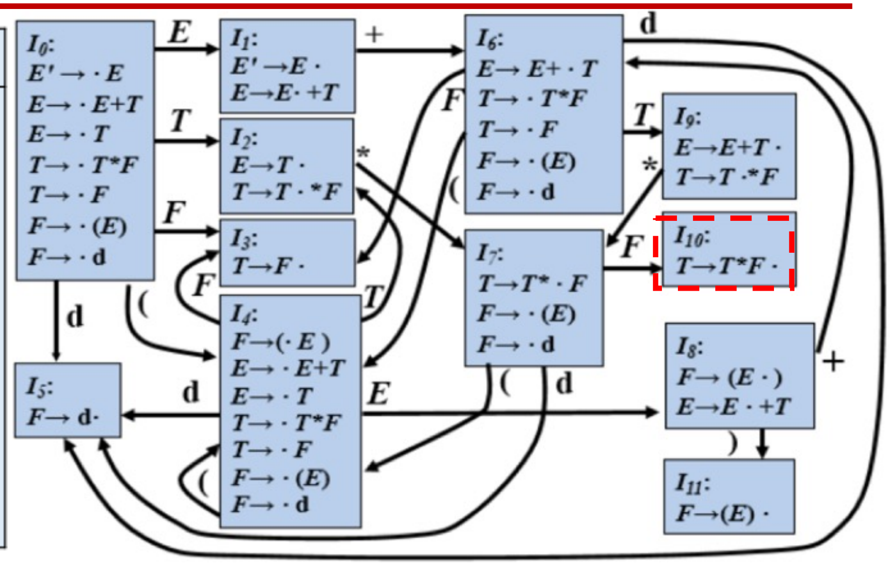
Input: 3 * 5 + 4

state →	S_0	S_3	S_7	S_{10}
symbol →	$	T	*	F
attribute →	-	3	-	5

https://www.icourse163.org/learn/HIT-1002123007

Example (cont.)

Productions	Semantic Actions
(1) L -> E	{ print(stack[top].val); }
(2) E -> E_1+T	{ stack[top-2].*val* = stack[top-2].*val* + stack[top].*val*; top = top -2; }
(3) E -> T	
(4) T -> T_1*F	{ stack[top-2].*val* = stack[top-2].*val* x stack[top].*val*; top = top -2; }
(5) T -> F	
(6) F -> (E)	{ stack[top-2].*val* = stack[top-1].*val*; top = top -2; }
(7) F -> digit	



Input: 3 * 5 + 4

state →	S_0	S_2	S_7	S_{10}
symbol →	$	T	*	F
attribute →	-	3	-	5
				top

state →	S_0	S_2
symbol →	$	T
attribute →	-	15
		top