中山大學 SUN YAT-SEN UNIVERSITY
计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第16讲：语义分析(2)

张献伟

xianweiz.github.io

DCS290, 4/25/2024

Review Questions

- Is it possible to perform semantic analysis during parsing?

  Yes. Syntax-directed translation, only one-pass.

- How to augment CFG to enable syntax directed?

  Symbol: attributes, production: semantic rule.

- SDD vs. SDT?

  SDD: syntax directed definition; SDT: translation scheme.

- Categories of semantic attributes?

  Synthesized (child, itself), inherited (parent, sibling, itself).

- Suppose X.x = A.a + Y.y, x is synthesized or inherited?

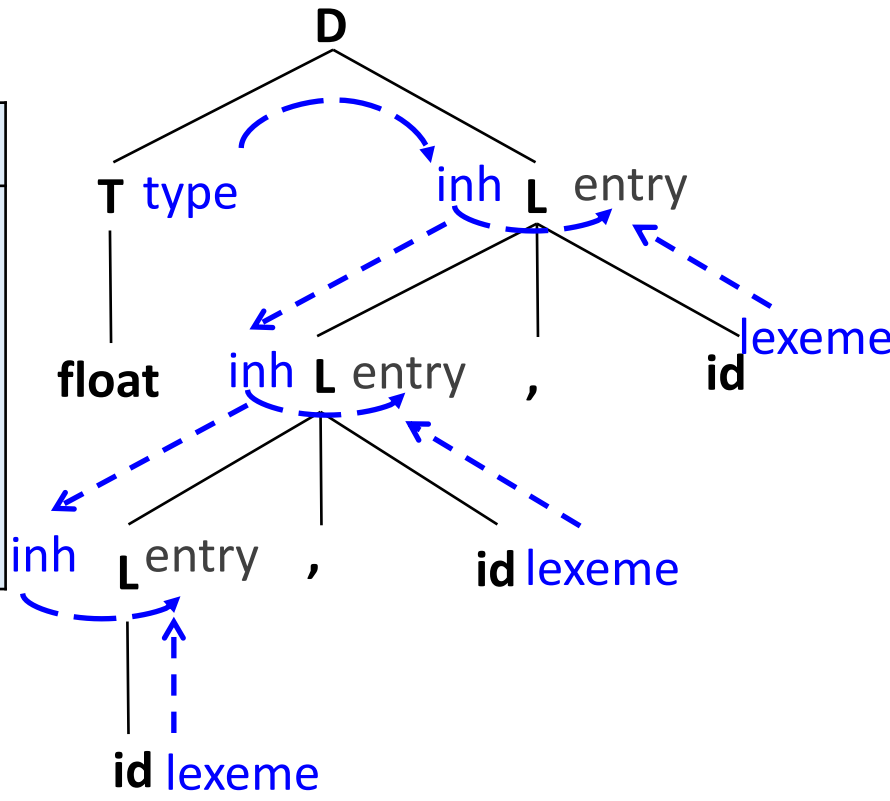
  Inherited (parent, sibling).

A -> X Y Z

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()*

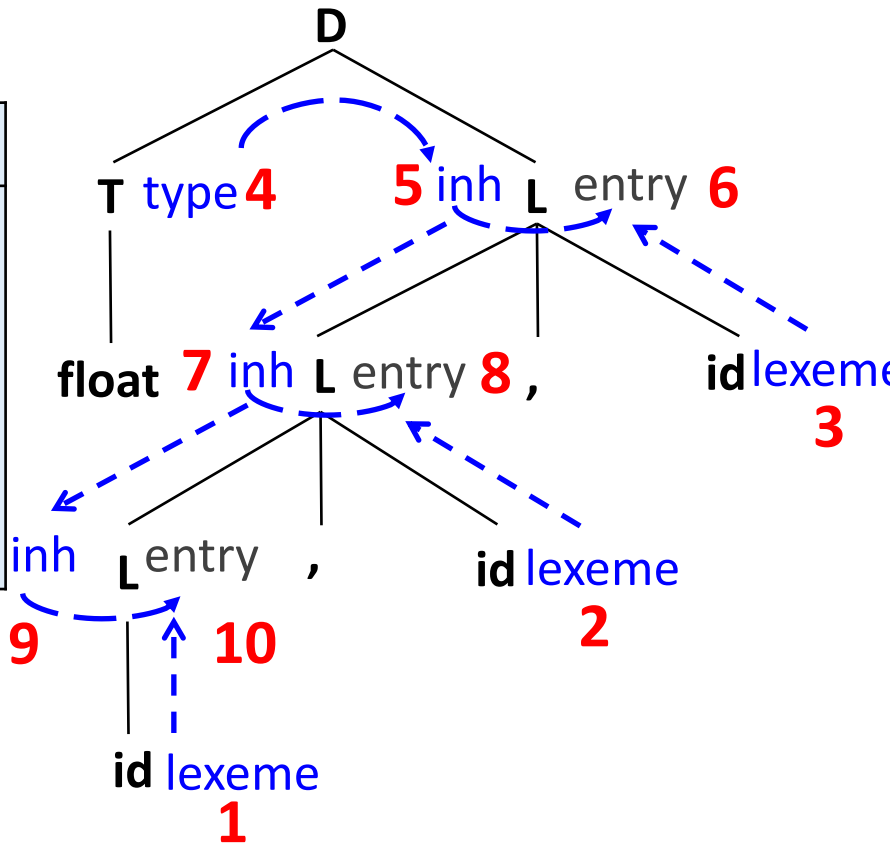
Evaluation Order[属性值计算顺序]

- 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

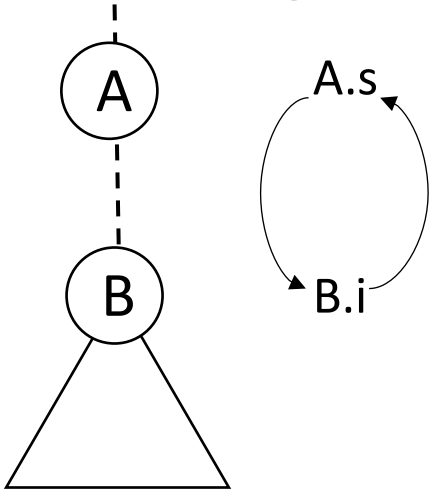
Topological sort:

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[依赖关系简单]
    - 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[依赖关系复杂]
- 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[一些SDD确保无循环]
    - 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（无循环、处理顺序）]
  - More precisely: each attribute must be either **synthesized**, or **inherited** but with the rules limited as follows: suppose A -> $X_1X_2...X_n$, the inherited attribute $X_i.a$ only depends on
    - **Inherited** attributes associated with A
    - Either ***syn* or *inh*** attributes of X_1, X_2, …, 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

SDD vs. SDT

- SDD[语法制导定义]: 是CFG的推广，翻译的高层次规则说明
  - A CFG grammar together with attributes and semantic rules
    - A subset of them are also called **attribute grammars**[属性文法]
      - No side effects, i.e., rules are strictly local to each production
  - Semantic rules imply no order to attribute evaluation
- SDT[语法制导翻译方案]: SDD的补充，具体翻译实施方案
  - An executable specification of the SDD
    - Fragments of progs are attached to different points in the production rules
  - The order of execution is important

Grammar	SDD	SDT
D -> T L	L.*inh* = T.*type*	D -> T { L.*inh* = T.*type* } L
T -> int	T.*type* = int	T -> int { T.*type* = int }
T -> float	T.*type* = float	T -> float { T.*type* = float }
L -> L_1, id	L_1.*inh* = L.*inh*	L -> { L_1.*inh* = L.*inh* }L_1, id
L -> id	id.*type* = L.*inh*	L -> { id.*type* = L.*inh*} id

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[一定按parse顺序]

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[S-SDD到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.)

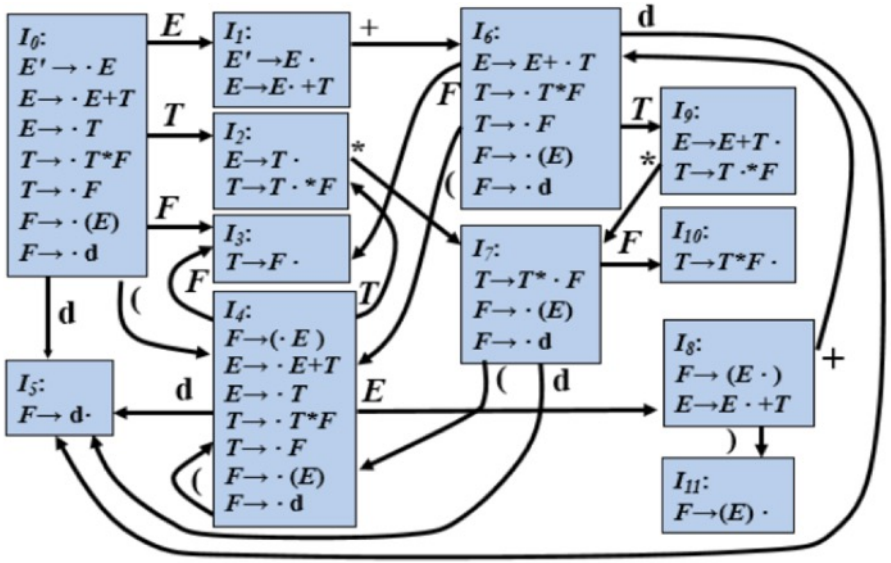
- If the underlying grammar of S-SDD is LR parsable
  - Then the SDT can be implemented during LR parsing
- Implement the converted SDT by[借助归约实现]
  - Executing the action along with the reduction of *head <- body*

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

symbol → $

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

symbol → $

attribute → -

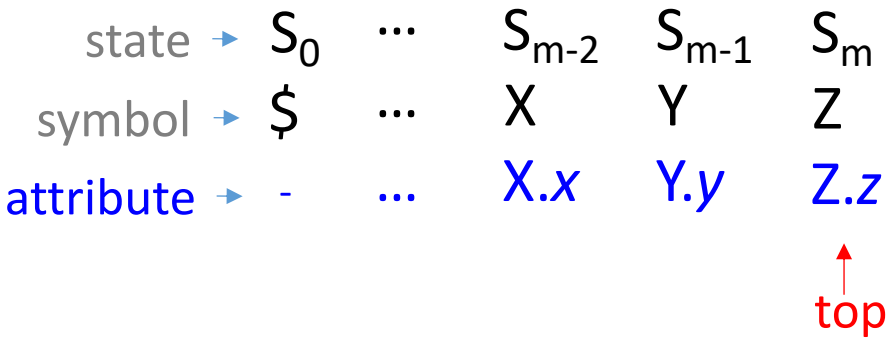
Extend LR Parse Stack[扩展分析栈]

- 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*

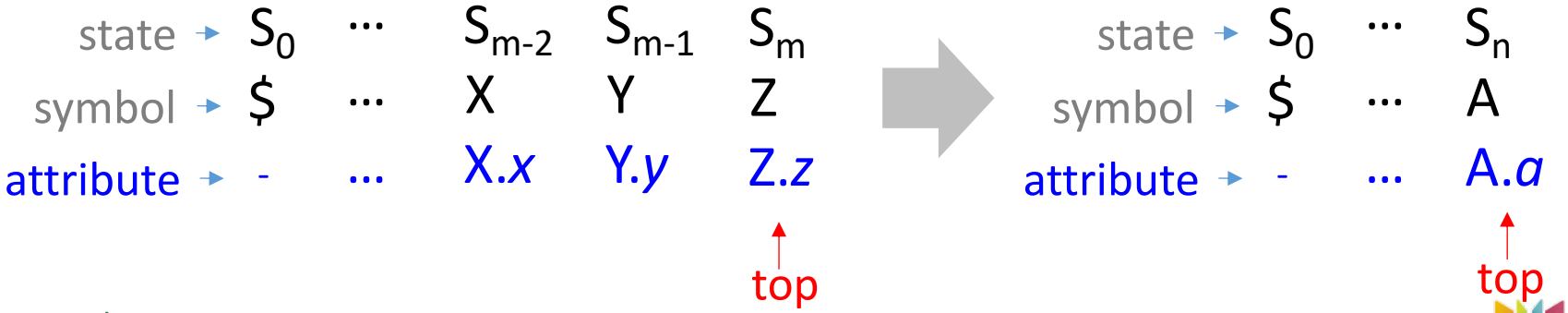
Extend LR Parse Stack[扩展分析栈]

- 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)



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)

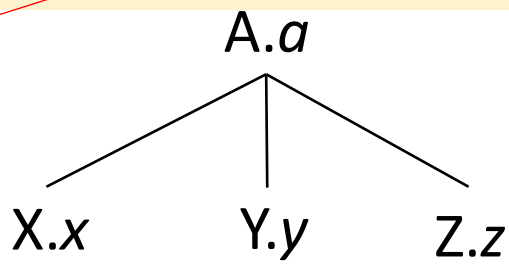


Stack Manipulation[栈操作]

- 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

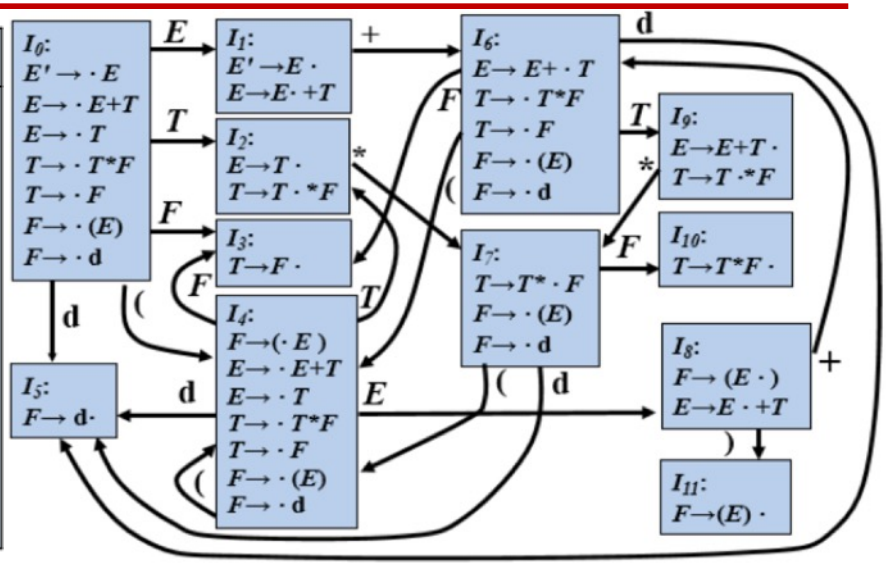
Example

- 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	



SLR Automaton

Input: 3 * 5 + 4

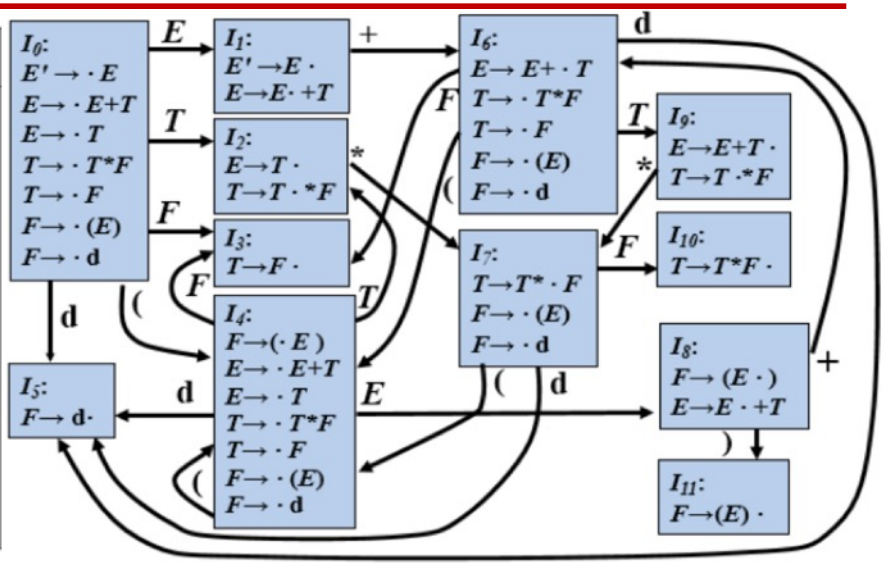
state → S_0

symbol → $

attribute → -

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	



SLR Automaton

Input: 3 * 5 + 4

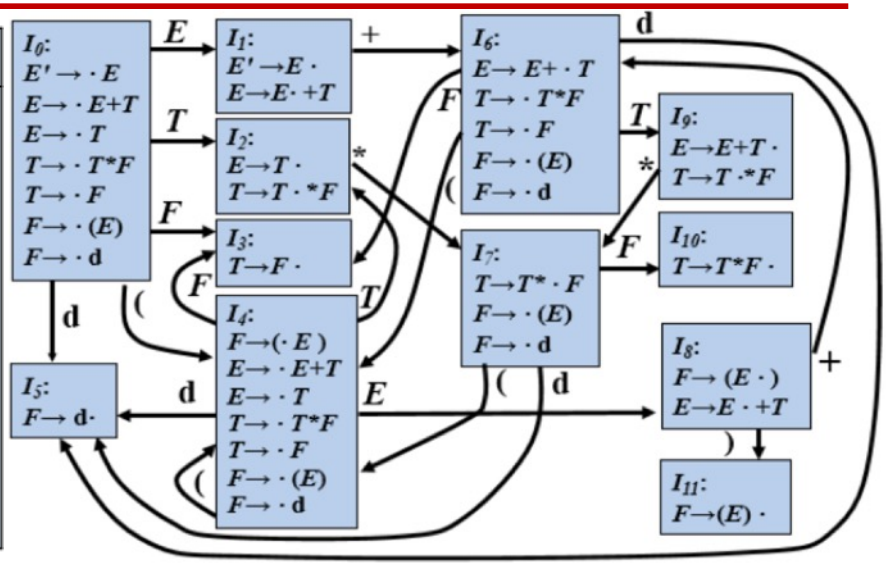
state → S_0

symbol → $

attribute → -

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	



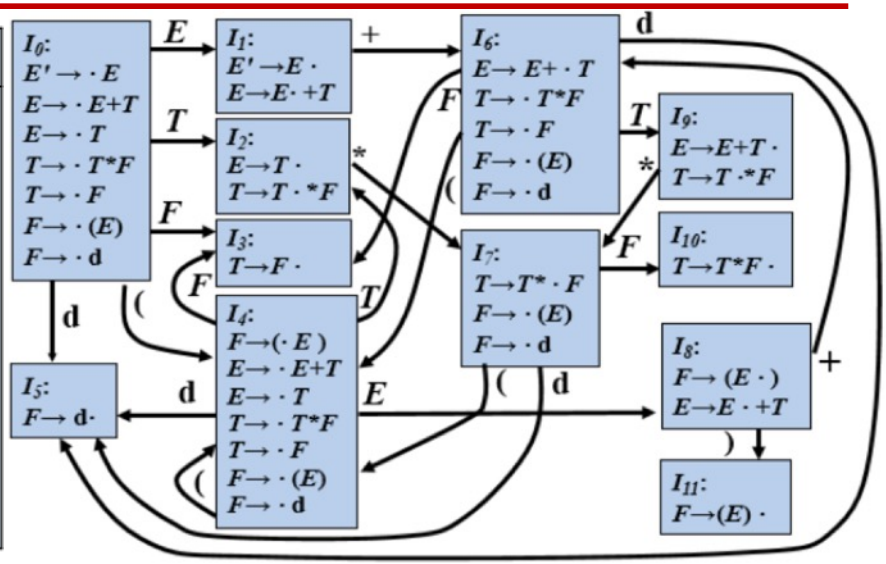
SLR Automaton

Input: 3 * 5 + 4

state →	S_0	S_5
symbol →	$	d
attribute →	-	3

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	



SLR Automaton

Input: 3 * 5 + 4

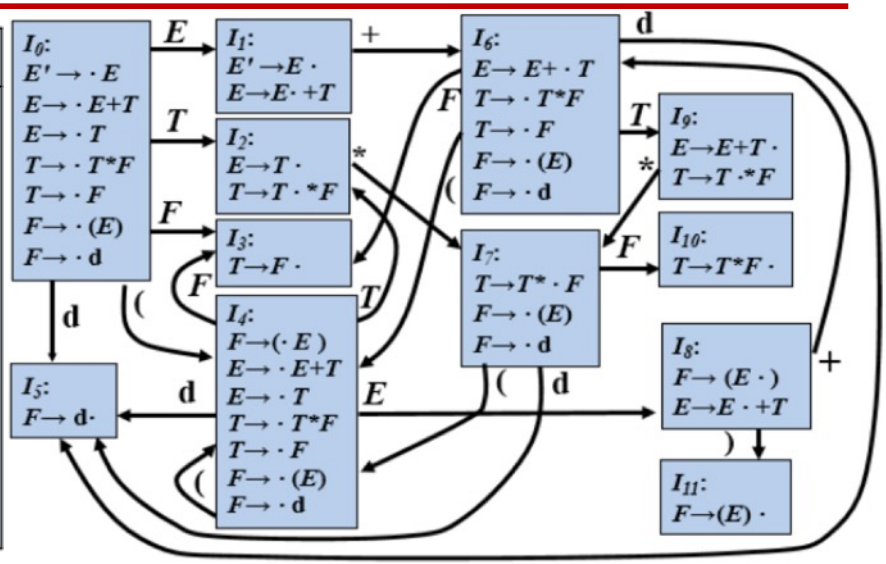
state → S_0

symbol → $

attribute → - 3

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	



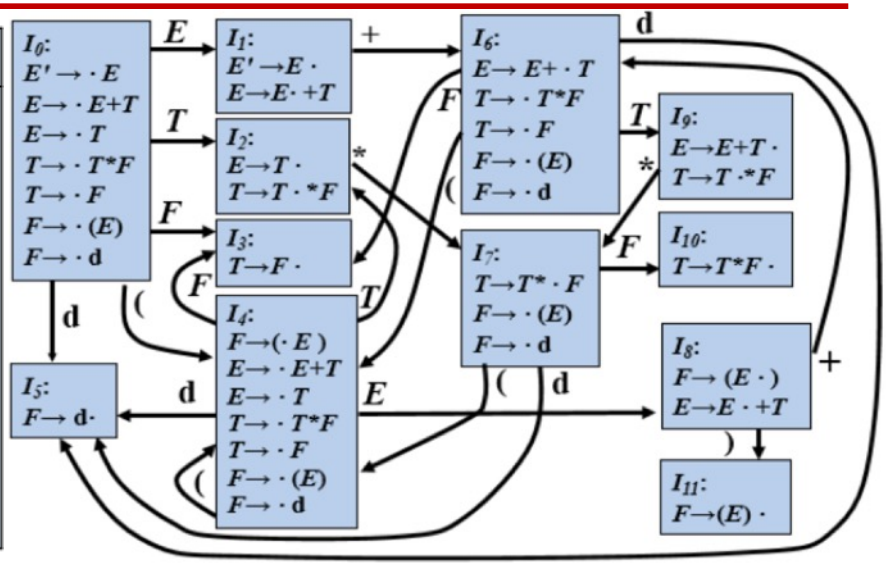
SLR Automaton

Input: 3 * 5 + 4

state →	S_0	S_3
symbol →	$	F
attribute →	-	3

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	



SLR Automaton

Input: 3 * 5 + 4

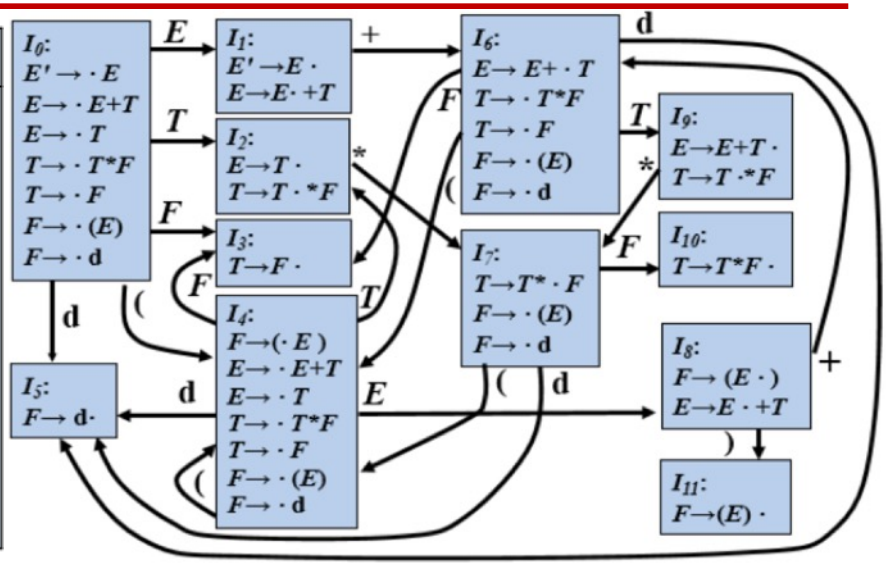
state → S_0

symbol → $

attribute → - 3

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	



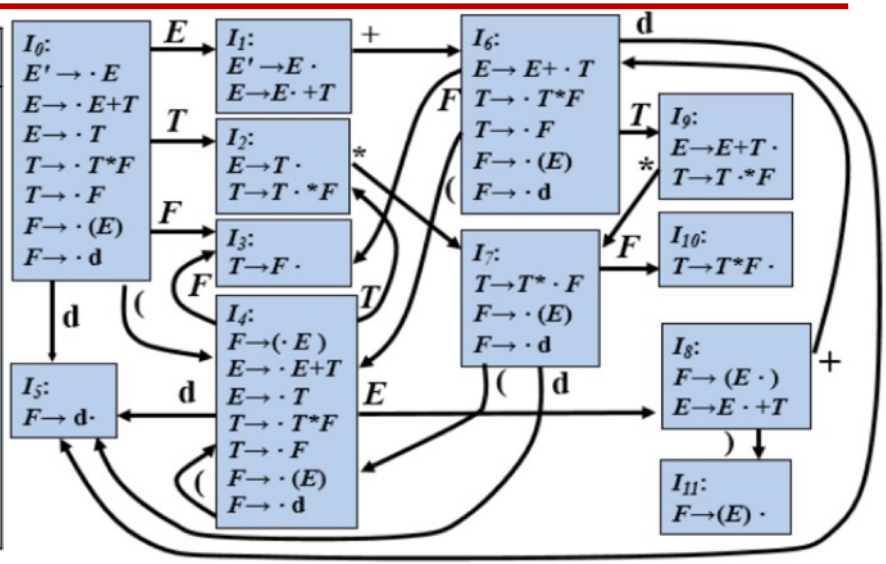
SLR Automaton

Input: 3 * 5 + 4

state	S_0	S_2
symbol	$	T
attribute	-	3

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	



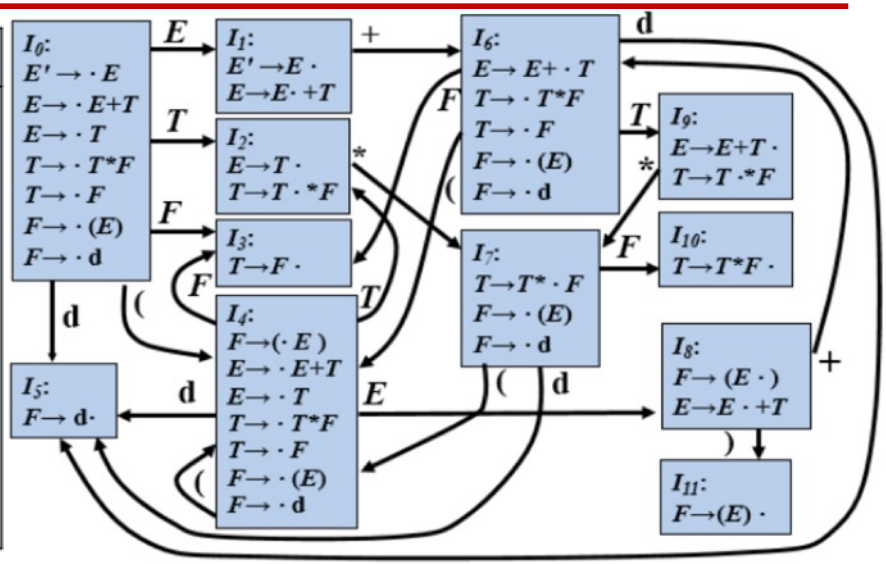
SLR Automaton

Input: 3 * 5 + 4

state	S_0	S_2
symbol	$	T
attribute	-	3

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	



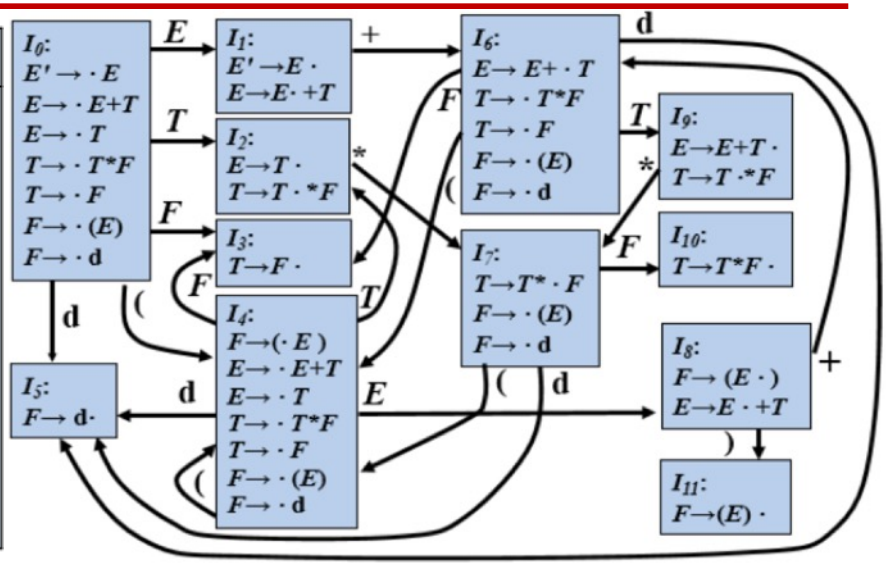
SLR Automaton

Input: 3 * 5 + 4

state →	S_0	S_2
symbol →	$	T
attribute →	-	3

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	



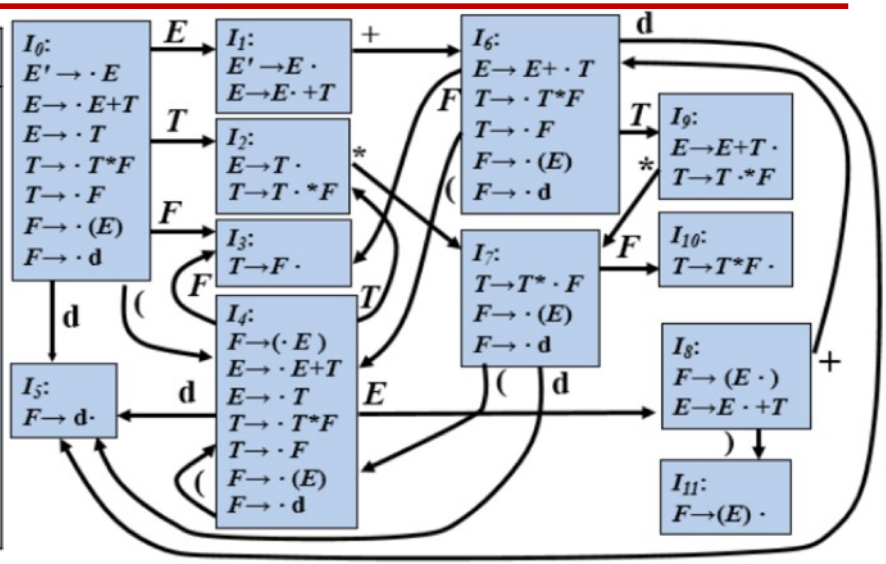
SLR Automaton

Input: 3 * 5 + 4

state	S_0	S_2	S_7
symbol	$	T	*
attribute	-	3	-

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	



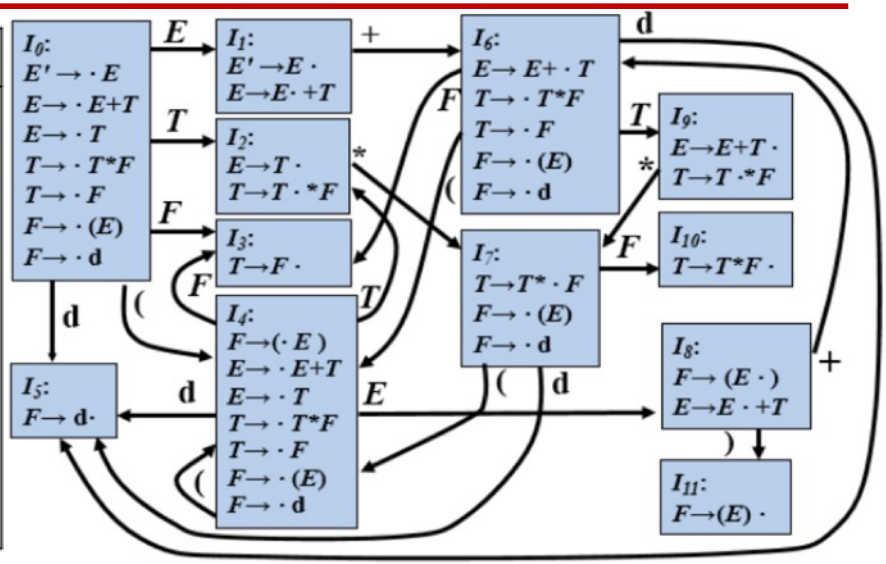
SLR Automaton

Input: 3 * 5 + 4

state	S_0	S_2	S_7
symbol	$	T	*
attribute	-	3	-

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	



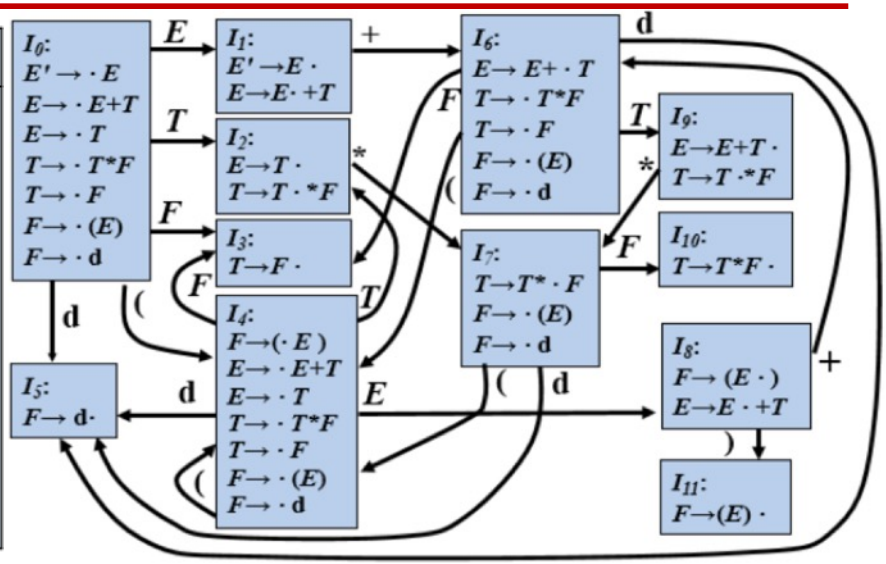
SLR Automaton

Input: 3 * 5 + 4

state →	S_0	S_2	S_7
symbol →	$	T	*
attribute →	-	3	-

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	



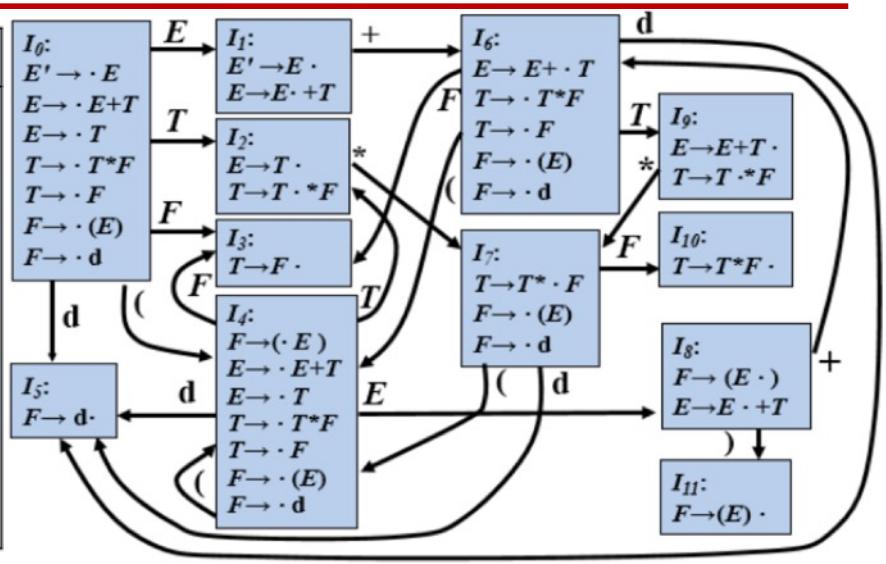
SLR Automaton

Input: 3 * 5 + 4

state →	S_0	S_2	S_7	S_5
symbol →	$	T	*	d
attribute →	-	3	-	5

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	



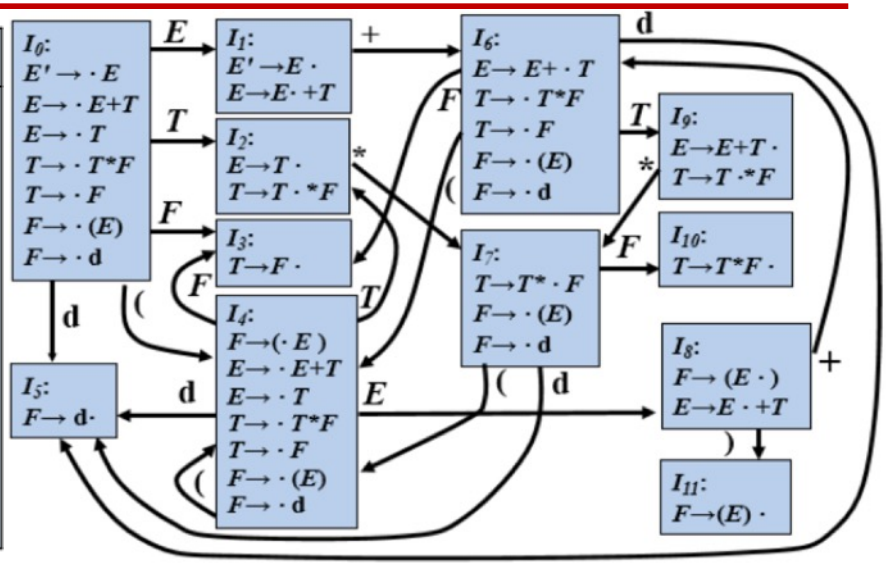
SLR Automaton

Input: 3 * 5 + 4

state →	S_0	S_2	S_7
symbol →	$	T	*
attribute →	-	3	-

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	



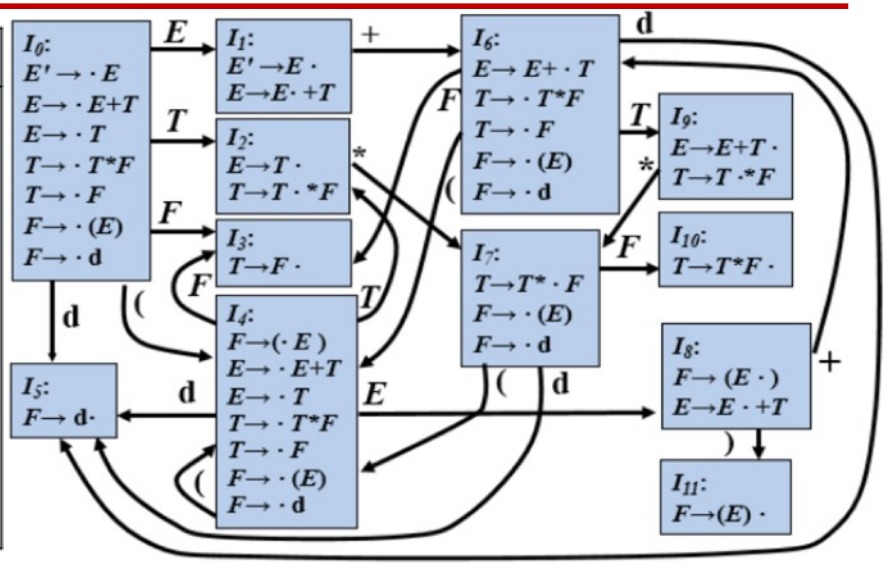
SLR Automaton

Input: 3 * 5 + 4

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

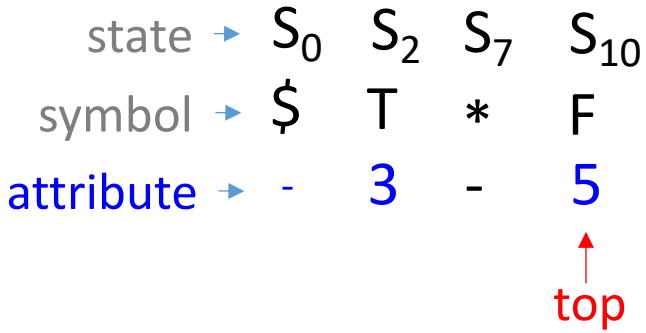
Example (cont.)

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	



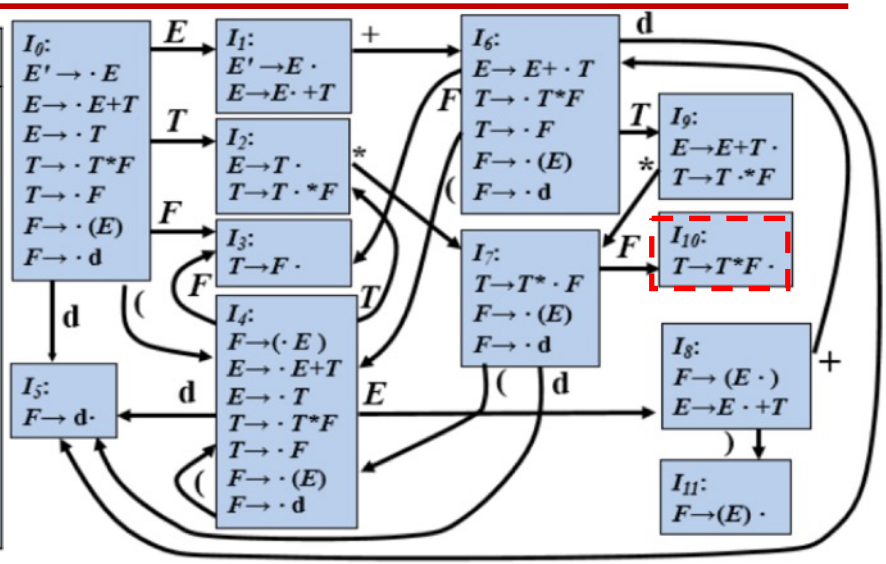
SLR Automaton

Input: 3 * 5 + 4



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	



SLR Automaton

Input: 3 * 5 + 4

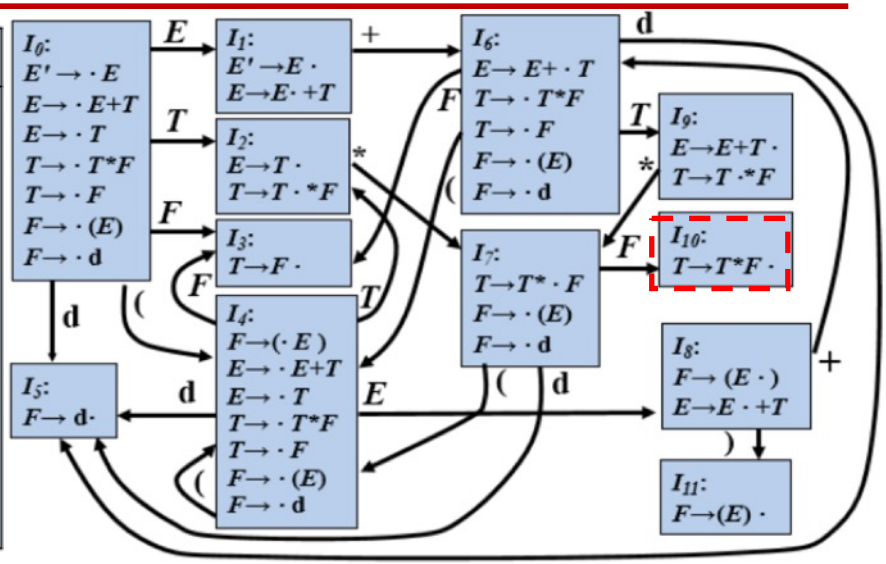
state →	S_0	S_2	S_7	S_{10}
symbol →	$	T	*	F
attribute →	-	3	-	5

top

state →	S_0
symbol →	$
attribute →	-

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	



SLR Automaton

Input: 3 * 5 + 4

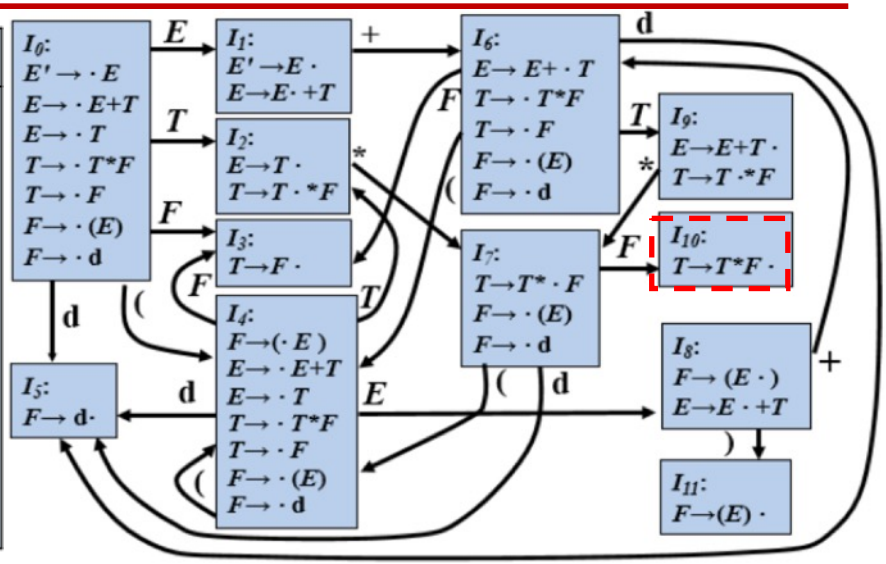
state →	S_0	S_2	S_7	S_{10}
symbol →	$	T	*	F
attribute →	-	3	-	5

top

state →	S_0	
symbol →	$	
attribute →	-	15

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	



SLR Automaton

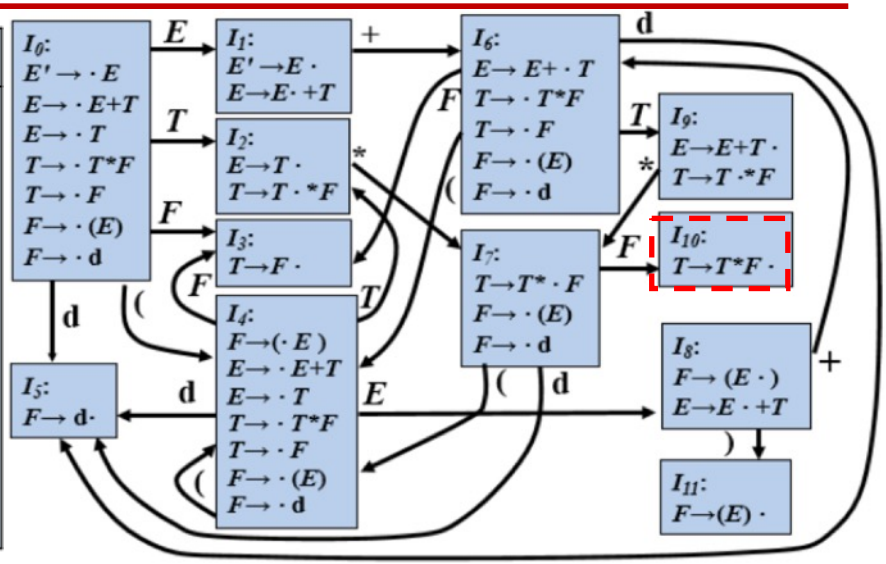
Input: 3 * 5 + 4

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

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

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	



SLR Automaton

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

== Implement L-SDD ==

- We have examined S-SDD -> SDT -> implementation
  - S-SDD can be converted to SDT with actions at production end
  - The SDT can be parsed and translated bottom-up, as long as the underlying grammar is LR-parsable
- What about the more general **L-SDD**?[L-属性文法]
  - Rules for turning L-SDD into an SDT
    - Embed the semantic rule that computes the inherited attributes for a nonterminal *A* immediately before that occurrence of *A* in the production body 综合属性呢 ??? 综合属性在A→xxx那个产生式处理！

      [将计算某个非终结符A的继承属性的语义规则插入到产生式右部中紧靠在A的本次出现之前的位置上] 能延后放吗？ A后的其他符号很可能依赖于A的属性！
    - Place the rules that compute a synthesized attribute for the head of a production at the end of the body of that production

      [将计算一个产生式左部符号的综合属性的规则放在这个产生式右部的末尾]

Example

A -> B C

- C的继承属性：出现之前
- A的综合属性：产生式末尾

Production Rules	Semantic Rules
(1) T -> F T’	T’.*inh* = F.*val*
	T.*val* = T’.*syn*
(2) T’ -> * F T_1’	T_1’.*inh* = T’.*inh* x F.*val*
	T’.*syn* = T_1’.*syn*
(3) T’ -> ε	T’.*syn* = T’.*inh*
(4) F -> digit	F.*val* = digit.*lexval*

Example

A -> B C

- C的继承属性：出现之前
- A的综合属性：产生式末尾

Production Rules	Semantic Rules
(1) T -> F T’	T’.*inh* = F.*val*
	T.*val* = T’.*syn*
(2) T’ -> * F T_1’	T_1’.*inh* = T’.*inh* x F.*val*
	T’.*syn* = T_1’.*syn*
(3) T’ -> ε	T’.*syn* = T’.*inh*
(4) F -> digit	F.*val* = digit.*lexval*

SDT

(1) T -> F { T’.*inh* = F.*val* } T’ { T.*val* = T’.*syn* }

(2) T’ -> * F { T_1’.*inh* = T’.*inh* x F.*val* } T_1’ { T’.*syn* = T_1’.*syn* }

(3) T’ -> ε { T’.*syn* = T’.*inh* }

(4) F -> digit { F.*val* = digit.*lexval* }

Implement the SDT of L-SDD

- If the underlying grammar is LL-parsable, then the SDT can be implemented during LL or LR parsing[若文法是LL可解析的，则可在LL或LR语法分析过程中实现]

- Semantic translation during **LL parsing**, using[LL方式]
  - A recursive-descent parser[LL递归下降]
    - Augment non-terminal functions to both parse and handle attributes
  - A predictive parser[LL非递归的预测分析]
    - Extend the parse stack to hold actions and certain data items needed for attribute evaluation
- A LR parser[LR方式]
  - Involve marker to rewrite grammars

L-SDD in LL Parsing[非递归预测]

- Extend the parse stack to hold **records** and certain **data items** needed for attribute evaluation[扩展语法分析栈]
  - Action-record[动作记录]: represent the actions to be executed
  - Synthesize-record[综合记录]: hold synthesized attrs for non-terminals
  - Typically, the data items are copies of attributes[属性备份]
- Manage attributes on the stack[管理属性信息]
  - The inherited attributes of a nonterminal *A* are placed in the stack record that represents that terminal[符号位放继承属性]
    - Action-record to evaluate these attributes are immediately above *A*
  - The synthesized attributes of a nonterminal *A* are placed in a separate synthesize-record that is immediately below *A*[综合属性另存放]

action	Code
A	Inh Attr.
A.syn	Syn Attr.

L-SDD in LL Parsing (cont.)

- Table-driven LL-parser
  - Mimics a leftmost derivation --> stack expansion
- *A* -> *BC*, suppose nonterminal *C* has an inherited attr *C.i*
  - *C.i* may depend not only on the inherited attr. of A, but on all the attrs of *B*[依赖于左侧]
    - Extra care should be taken on the attribute values
  - Since SDD is L-attributed, surely that the values of the inherited attrs of *A* are available when *A* rises to stack top ($X \rightarrow \alpha A \beta$)[左侧就绪]
    - Thus, available to be copied into C
  - *A*'s synthesized attrs remain on the stack, below *B* and *C* when expansion happens[综合属性也还在栈里]

action	Code
A	Inh Attr.
A.syn	Syn Attr.

L-SDD in LL Parsing (cont.)

- *A* -> *BC*: *C.i* may depend not only on the inherited attr. of *A*, but on all the attrs of *B*
  - Thus, need to process *B* completely before *C.i* can be evaluated
  - Save temporary copies of all attrs needed by evaluate *C.i* in the action-record that evaluates *C.i*; otherwise, when the parser replaces *A* on top of the stack by *BC*, the inherited attrs of *A* will be gone, along with its stack record
  - 变量展开时（i.e., 变量本身的记录出栈时），若其含有继承属性，则要将继承属性复制给后面的动作记录
  - 综合记录出栈时，要将综合属性值复制给后面的动作记录

action	Code
A	Inh Attr.
A.syn	Syn Attr.

Example

(1) T -> F { T'.*inh* = F.*val* } T' { T.*val* = T'.*syn* }

(2) T' -> * F { T_1'.*inh* = T'.*inh* x F.*val* } T_1' { T'.*syn* = T_1'.*syn* }

(3) T' -> ε { T'.*syn* = T'.*inh* }
(4) F -> digit { F.*val* = digit.*lexval* }

Three kinds of symbols:
1) terminal
2) non-terminal
3) action

(1) T -> F { a_1 } T' { a_2 }	a_1: T'.*inh* = F.*val*
	a_2: T.*val* = T'.*syn*
(2) T' -> * F { a_3 } T_1' { a_4 }	a_3: T_1'.*inh* = T'.*inh* x F.*val*
	a_4: T'.*syn* = T_1'.*syn*
(3) T' -> ε { a_5 }	a_5: T'.*syn* = T'.*inh*
(4) F -> digit { a_6 }	a_6: F.*val* = digit.*lexval*