

# 编译原理 Complier Principles

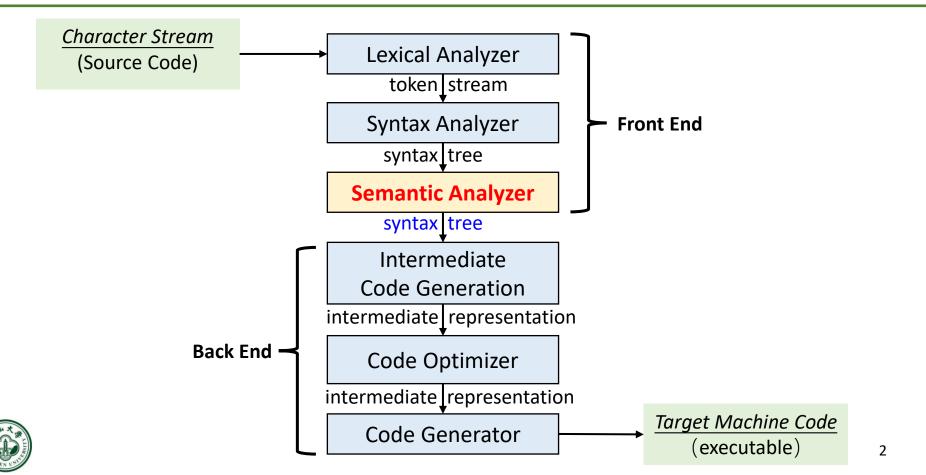
# Lecture 6 Semantic Analysis: Intro & SDD & SDT

赵帅

计算机学院 中山大学

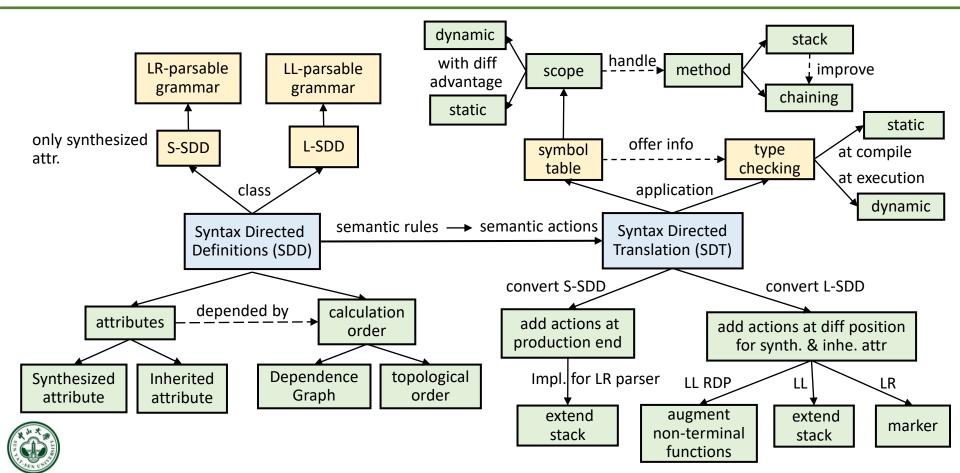
### Compilation Phases[编译阶段]





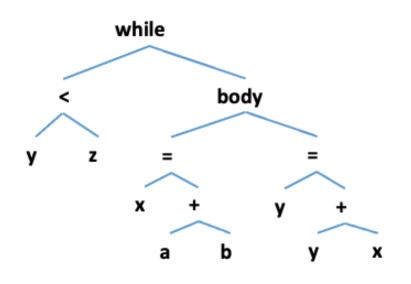
#### Content



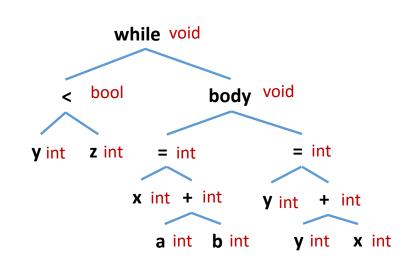


### Compilation Phases[编译阶段]





Abstract Syntax Tree(AST)



Annotated AST/Decorated AST [带标注的抽象语法树]



# Why Semantic Analysis? [语义分析]



- Because programs use symbols (a.k.a. identifiers)
  - ◆ Identifiers require context to figure out the meaning
- Consider the English sentence: "He ate it"
  - ◆ This sentence is syntactically correct
  - ◆ But it makes sense only in the context of a previous sentence: "Sam bought a pizza." (what if "Sam bought a car."?)

#### Semantic analysis

◆ Associates identifiers with objects they refer to[关联]

```
□ "He" --> "Sam"
```

◆ Checks whether identifiers are used correctly[检查]

"He" and "it" refer to some object: def-use check

"it" is a type of object that can be eaten: type check



### Why Semantic Analysis?



- Semantic of a language is more difficult to describe than syntax [语义比语法更难描述]
  - ◆ Syntax: describes the proper form of the programs [仅形式]
  - ◆ <u>Semantics</u>: defines the meaning of programs (i.e., what each program does when it executes) [到意义]
- Context cannot be analyzed using a CFG parser[cfg不能分析上下文信息]
  - Associating IDs with objects require expressing the pattern:

```
\{wcw \mid w \in (a \mid b)^*\}
```

- The first w represents the definition of an ID
- □ The c represents arbitrary intervening code
- □ The second w represents the use of the ID



### **Semantic Analysis**



- Deeper check into the source program[对程序进一步分析]
  - ◆ Last stage of the front end[前端最后阶段]
  - ◆ Compiler's <u>last chance</u> to reject incorrect programs[最后拒绝机会]
  - Verify properties that aren't caught in earlier phases
    - □ Variables declaration before use [先声明后使用]
    - □ Type consistency when using IDs [变量类型一致]
    - □ Correct expressions types [表达式类型]
- Gather useful info about program for later phases[收集后续信息]
  - Determine what variables are meant by each identifier
  - Build an internal representation of inheritance hierarchies
  - ◆ Count how many variables are in scope at each point



### **Semantic Analysis: Implementation**

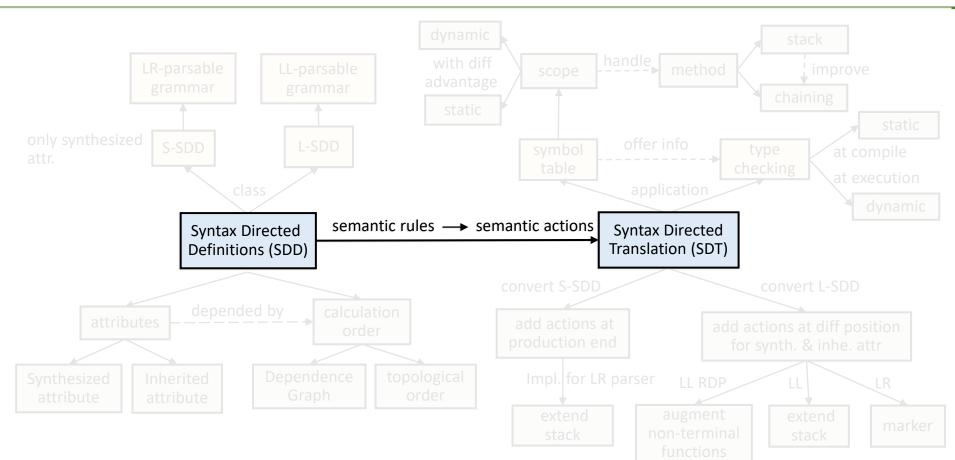


- Attribute grammars[属性文法]
  - One-pass compilation
    - Semantic analysis is done along with parsing
  - Augment rules to do checking during parsing
  - ◆ The approach suggested in the Compilers book
- AST walk[语法树遍历]
  - ◆ Two-pass compilation
    - □ First pass digests the syntax and builds a parse tree
    - The second pass traverses the tree to verify that the program respects all semantic rules
  - ◆ Strict phase separation of Syntax Analysis and Semantic Analysis



#### Content

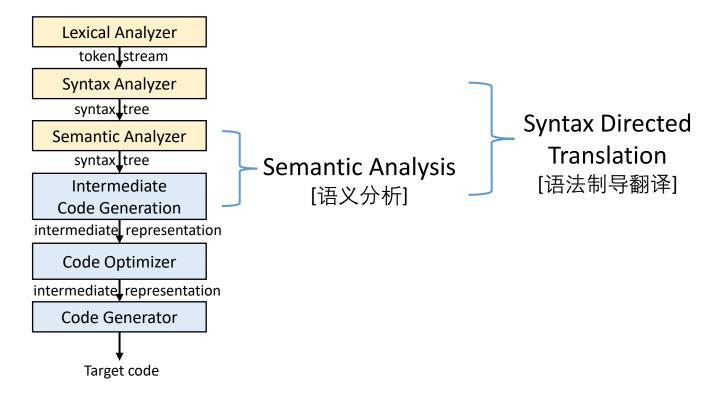




### **Syntax Directed Translation**



• Syntax Directed Translation[语法制导翻译]





#### Syntax Directed Translation[语法制导翻译]





- Translate based on the program's grammar structure[语法结构]
  - ◆ Syntactic structure: structure of a program given by grammar[文法]
  - ◆ The parsing process and parse trees are used to direct semantic analysis and the translation of the program, i.e., **CFG-driven translation**[CFG驱动的翻译]
- How? Augment the grammar used in parser[对语法的进一步加强]:
  - ◆ Attach semantic attributes[语义属性] to each grammar symbol
    - □ The attributes describe the symbol properties [符号特征]
    - An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register...
  - ◆ For each grammar production, give semantic rules or actions[语义规则或动作]
    - The actions describe how to compute the attribute values associated with each symbol in a production



#### Attributes[语义属性]



- Attributes can represent anything depending on the task[属性可以表示任意含义]
  - ◆ If computing expression: a number (value of expression)
  - ◆ If building AST: a pointer (pointer to AST for expression)
  - If generating code: a string (assembly code for expression)
  - If type checking: a type (type for expression)
- Attributes are associated directly with the grammar symbols

• Format: X.a (X is a symbol, a is one of its attributes)



#### SDD & SDT



Associating semantic rules with grammar rules (productions) involves two concepts:

- Syntax Directed Definitions (SDD) [语法制导定义]
- Syntax Directed Translation scheme (SDT) [语法制导翻译方案]



# Syntax Directed Definitions (SDD)





- Syntax Directed Definitions (SDD) [语法制导定义]
  - ◆ A syntax-directed definition (SDD) is a context-free grammar(CFG) together with attributes and rules
  - ◆ Add Attributes + semantic rules[语义规则] in CFG
  - ◆ A generalization of CFG
    - □ Attributes for grammar symbols [文法符号和语义属性关联]
    - Semantic rules for productions [产生式和语义规则关联]

Productions	Attributes	Semantic rules
E -> E1 + E2	E.val; E1.val; E2.val	E.val = E1.val + E2.val
E -> id	E.val; id.lexval	E.val = id.lexval



# **Syntax Directed Translation (SDT)**



- Syntax Directed Translation scheme (SDT) [语法制导翻译方案]
  - ◆ SDT is a CFG with program fragments embedded in the right part of the production, and these program fragments are called **semantic actions**.
  - ◆ Attributes + semantic actions[语义动作]
  - ◆ Example actions for computing the value of an expression

```
E -> E1 + E2 {E.val = E1.val + E2.val}
E -> id {E.val = id.lexval}
...
```

The position of a semantic action in a production determines the execution point of the action



#### SDD v.s. SDT



- SDD[语法制导定义]: 是CFG的推广, 翻译的高层次规则说明
  - ◆ A <u>CFG grammar</u> together with <u>attributes</u> and <u>semantic rules</u>
    - ◆ A subset of them are also called attribute grammars[属性文法]
  - ◆ 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
    - Program fragments are attached to different points in production rules
  - ◆ The execution order is important

D -> TL

T -> int

T -> float

L -> L<sub>1</sub>, id

#### SDD

L.inh = T.type

T.type = int

T.type = float

L1.inh = L.inh

#### SDT

D ->T {L.inh = T.type} L

T -> int {T.type = int}

T -> float {T.type = float}

 $L \rightarrow \{L1.inh = L.inh\} L_1$ , id



#### SDD v.s. SDT



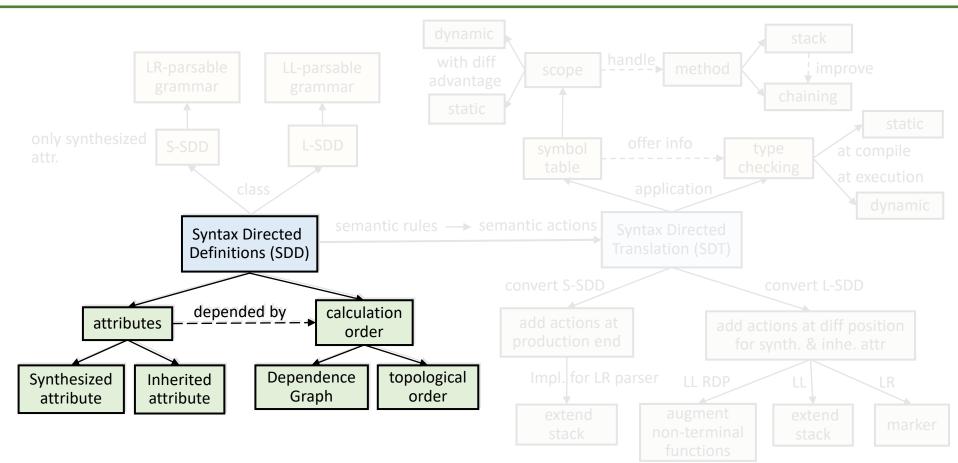
- Syntax: A ->  $\alpha$  {action<sub>1</sub>}  $\beta$  {action<sub>2</sub>}  $\gamma$  ...
- Actions are executed "at that point" in the RHS
  - action<sub>1</sub> executes after  $\alpha$  has been produced but before  $\beta$
  - action<sub>2</sub> executes after  $\alpha$ , action<sub>1</sub>,  $\beta$  but before  $\gamma$
- Semantic rule v.s. Action[语义规则 vs. 语义动作]
  - ◆ Semantic rules are not associated with locations in RHS SDD doesn't impose any order other than dependences
  - Location of action in RHS specifies when it should occur SDT specifies the execution order and time of each action





#### Content





# **Syntax Directed Definitions (SDD)**



- SDD是CFG的增广
  - Grammar symbols together with semantic attributes
  - Productions associated with a set of semantic rules to compute the value of an expression

- SDD has two types of attributes [文法符号的两种属性]
  - ◆ Synthesized attribute [综合属性]
  - ◆ Inherited attribute [继承属性]



# **Syntax Directed Definitions (SDD)**



- Synthesized attribute[综合属性]
  - ◆ Defined by a semantic rule associated with the <u>production at N</u>
     □ The production must have A as its head (i.e., A -> ...)
  - ◆ A synthesized attribute of node N is defined only by attributes of N's children and N itself[子节点或自身]

- Inherited attribute[继承属性]
  - ◆ Defined by a semantic rule associated with the production at the parent of N
     □ The production must have A as a symbol in its body (i.e., ... -> ...A...)
  - ◆ An inherited attributed of node N is defined only by attribute values at N's parent, N itself, and N's siblings[父节点、自身或兄弟节点]

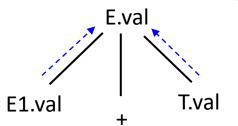


### Synthesized Attribute[综合属性]



- Synthesized attribute for non-terminal A of parse-tree node N[非终结符的综合属性]
  - ◆ Only defined by N's children and N itself
    - Passed up the tree
    - □ P.syn\_attr = f(P.attrs, C<sub>1</sub>.attrs, C<sub>2</sub>.attrs, C<sub>3</sub>.attrs)
  - ◆ Example

Productions	Semantic rules
E -> E1 + T	E.val = E1.val + T.val



- Terminals CAN have synthesized attributes[终结符可以具有综合属性]
  - Lexical values supplied by the lexical analysis
  - ◆ Thus, no semantic rules in SDD for terminals

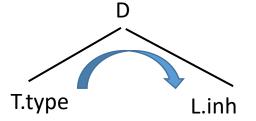


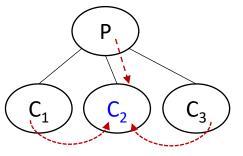
#### Inherited Attribute[继承属性]



- Inherited attribute for non-terminal A of parse-tree node N[非终结符继承属性]
  - ◆ Only defined by N's parent, N's siblings and N itself
    - □ Passed down a parse tree
    - $C_2.inh_attr = f(P.attrs, C_1.attrs, C_2.attrs, C_3.attrs)$
  - ◆ Example

Productions	Semantic rules
D -> T L	L.inh = T.type





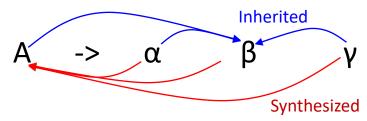
- Terminals CANNOT have inherited attributes[终结符无继承属性]
  - Only synthesized attributes from lexical analysis



#### SDD[语法制导定义]



• Attribute dependencies in a production rule[产生式中的属性依赖]



- SDD has rule of the form for each grammar production
  - b = f(A.attrs,  $\alpha$ .attrs,  $\beta$ .attrs,  $\gamma$ .attrs)
- b is either an attribute in LHS (an attribute of A)
  - ◆ In which case b is a **synthesized** attribute
  - $\bullet$  Why? From A's perspective  $\alpha$ ,  $\beta$ ,  $\gamma$  are children
- Or, b is an attribute in RHS (e.g., of β)
  - ◆ In which case b is an inherited attribute
  - Why? From  $\beta$ 's perspective A,  $\alpha$ ,  $\gamma$  are parent or siblings



### **Example: Synthesized attribute**



#### Attribute grammar for simple integer arithmetic expression

Grammar rule	Semantic Rules
$E \rightarrow E_1 + T$	E.val =E₁.val +T.val
E→T	E.val =T.val
T→T₁*F	T.val =T₁.val * F.val
T→F	T.val = F.val
F→(E)	F.val = E.val
F→dight	F.val = dight.val

The val attribute is synthesized



### **Example: Inherited Attribute**



#### Attribute grammar for variable

Grammar rule	Semantic Rules
decl -> type varlist	varlist.dtype = type.dtype
type -> int	type.dtype = integer
type -> float	type.dtype = real
varlist1 -> id, varlist2	id.dtype = varlist1.dtype
varlist -> id	varlist2.dtype = varlist1.dtype
	id.dtype = varlist.dtype

The dtype attribute is inherited



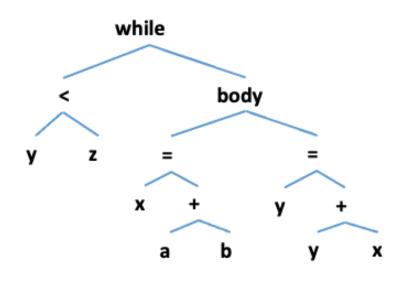
#### The Concepts



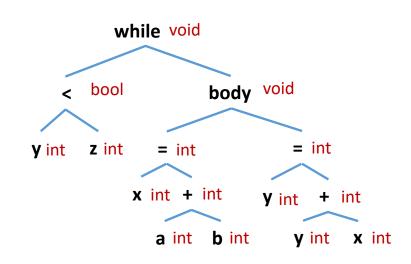
- Side effect[副作用]
  - ◆ 一般属性值计算(基于属性值或常量进行的)之外的功能
  - ◆ 例如: code generation, print results, modify symbol table ...
- Attribute grammar[属性文法]
  - ◆一个没有副作用的SDD
  - ◆The rules define the value of an attribute purely in terms of the value of other attributes and constants[属性文法的规则仅仅通过其他属性值和常量来定义一个属性值]
- 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]

### Compilation Phases[编译阶段]









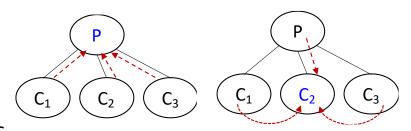
Annotated AST/Decorated AST [带标注的抽象语法树]



### **Semantic Analysis Revisit**



- SDD (Syntax-directed definition)
  - CFG + attributes for each symbol + semantic rules for each production
- SDT(Syntax-directed definition)
  - CFG + attributes for each symbol + {semantic actions} in each production
- synthesized attribute
  - Terminals CAN have synthesized attributes
- Inherited Attribute
  - Terminals **CANNOT** have inherited attributes



Grammar
D -> TL
T -> int
T -> float
L -> L<sub>1</sub>, id

SDD (rules)
L.inh = T.type
T.type = int
T.type = float
L1.inh = L.inh

SDT (actions)
D ->T {L.inh = T.type} L
T -> int {T.type = int}
T -> float {T.type = float}
L -> {L1.inh = L.inh} L<sub>1</sub>, id



#### **Dependence Graph**[依赖图]

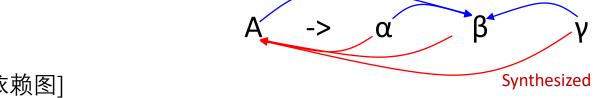


Inherited

• 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



#### Dependency graph[依赖图]

- ◆ While the annotated parse tree shows the values of attributes, a dependency graph helps determine <a href="how those values can be computed">how those values can be computed</a>[决定属性值计算]
- ◆ Depicts the flow of information among the attribute instances in a particular parse tree[描绘了分析树的属性信息流]
  - Edges are dependence relationships between attributes
  - □ For each parse-tree node X, there's a graph node for each attr of X
  - □ If X.a depends on Y.b, then there's one directed edge from X.a to Y.b



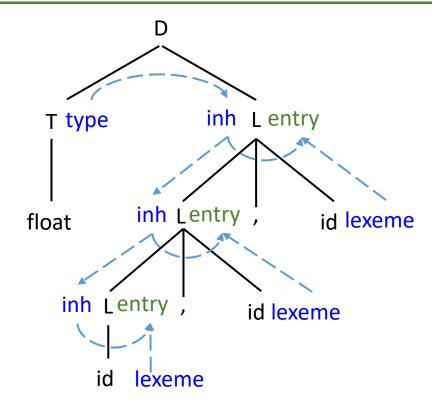
### **Example: Dependency Graph**



#### SDD:

Productions	Semantic rules
D -> T L	L.inh = T.type
T -> int	T.Type = int
T -> float	T.Type = float
L -> L <sub>1</sub> , id	L1.inh = L.inh
	addtype(id.entry,L.inh)
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[依赖图描述了属性计算顺序]
- If the graph has an edge <u>from</u> node M <u>to</u> node N, then the attribute associated with M must be <u>evaluated before</u> 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 from  $N_i$  to  $N_j$  and i < j
  - ◆ This order embeds a directed graph into a linear order, 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**

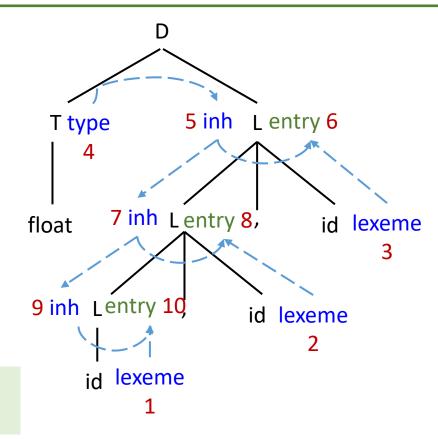


#### SDD:

Productions	Semantic rules
D -> T L	L.inh = T.type
T -> int	T.type= int
T -> float	T.type= float
L -> L <sub>1</sub> , id	L1.inh=L.inh
	addtype(id.entry,L.inh)
L-> id	addtype(id.entry,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
  - ◆ Inherited: evaluate parent and dependent siblings first, then the node itself
  - ◆ For SDD's with both inherited and synthesized attributes, there's no guarantee that there is even one evaluation order
- Difficult to check circularities in a dependency graph
  - But, there are subclasses of SDD's that guarantee an evaluation order
    - Such classes do not permit graphs with cycles

Production

Semantic Rules

A-> B

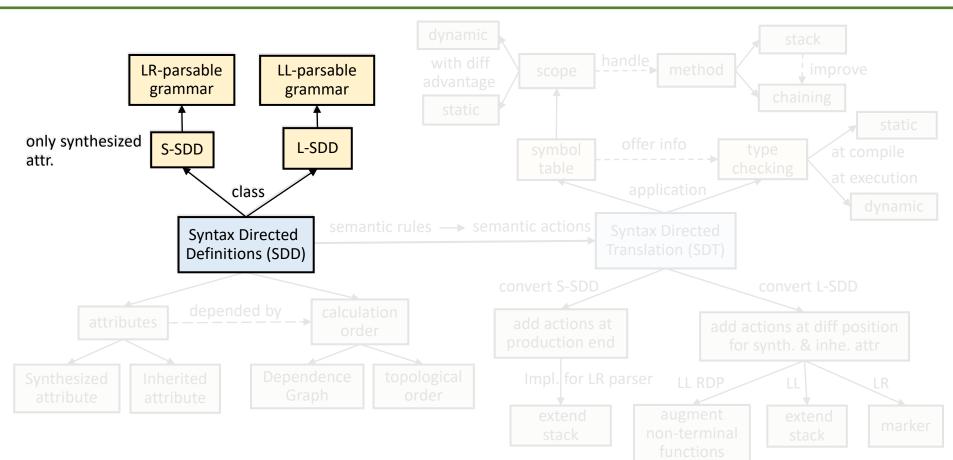
A.s=B.i;

B.i = A.s + 1;



#### Content





#### S-Attributed Definitions[S-属性定义]



- An SDD is **S-attributed** if every attribute is <u>synthesized</u>[只具有综合属性]
- If an SDD is S-attributed (S-SDD)
  - ◆ Can evaluate its attributes in any bottomup order of the nodes of the parse-tree[自底 向上的顺序计算属性值]
  - ◆ Can be implemented during bottom-up parsing[与自底向上的语法分析同时进行]

Productions	Semantic rules
(1)L-> E	print(E.val)
(2)E-> E <sub>1</sub> + T	$E.val = E_1.val + T.val$
(3)E-> T	E.val =T.val
(4)T-> T <sub>1</sub> * F	T.val= T <sub>1</sub> .valxF.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]

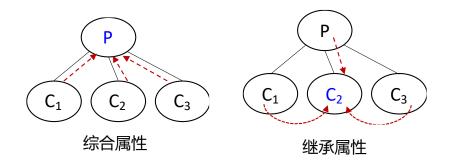


- An SDD is L-attributed (L-SDD) if
  - ◆ Between the attributes associated with a production body, dependency graph edges can only go from <u>left to right</u> [依赖图的边只能从左到右]
  - Each attribute can be either **synthesized**, or **inherited** with the rules that suppose  $A \rightarrow X_1X_2...X_{i-1}X_i$ , the inherited attribute  $X_i$  only depends on:
    - 1. Inherited attributes associated with A [A的继承属性, Why?]
    - 2. Either syn or inh attributes of X<sub>1</sub>, X<sub>2</sub>, ..., X<sub>i-1</sub> located to the left of X<sub>i</sub> [X<sub>i</sub>的之前符号的综合/继承属性]
    - 3. Either syn or inh attributes of  $X_i$  itself, but no cycles formed by the attributes of this  $X_i$  [ $X_i$ 的综合/继承属性 (不能出现环形依赖)]
  - ◆ L-SDD是对S-SDD的拓展, Can be implemented during <u>top-down/bottom-up</u> parsing [L属性的语义分析可在LL/LR分析中完成]



#### L-Attributed Definitions (cont.)





S-SDD or L-SDD?

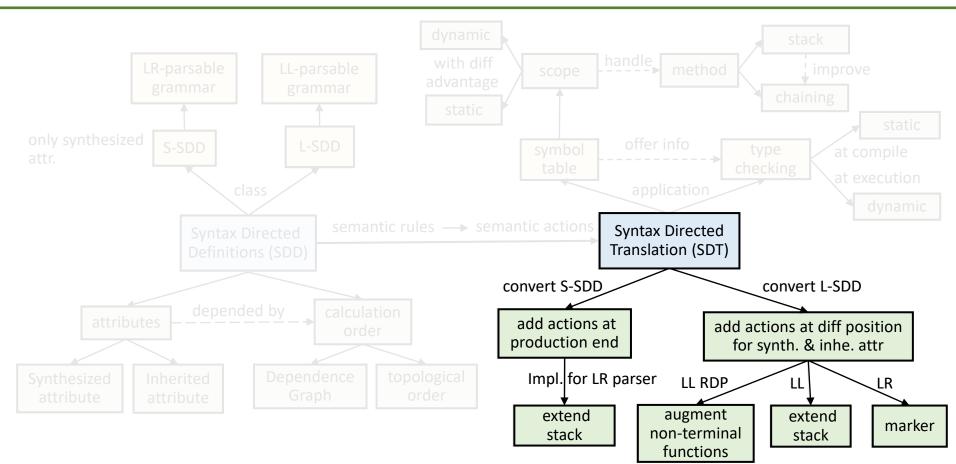
Productions	Semantic rules
A -> BC	A.s = B.b $B.l = f(C.c)A.s)$

Not S-SDD: B.i is inh [S属性文法中不会出现 继承属性] Not L-SDD: A.s is syn arrt, C is right to B [L属性文法中,继承属性 (B.I)不会依赖综合属性(A.s)] [L属性文法中, RHS不会出现从右到左的依赖关系]



#### Content





# Implementation[实现]



- SDT (executable SDD) can be implemented in two ways
  - ◆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
    - a 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



# **Implementation**[实现]



- Two important classes of SDD's[两个关键子类]
  - ◆ SDD is <u>S-attributed</u>, the underlying grammar is <u>LR-parsable</u>
  - ◆ SDD is <u>L-attributed</u>, the underlying grammar is <u>LL-parsable</u>

- 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[SDD到SDT的转换]
  - ◆ Place 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** [后缀/尾部SDT]

S-SDD

Productions	Semantic rules
(1)L-> E	print(E.val)
(2)E-> E <sub>1</sub> + T	E.val= E <sub>1</sub> .val+T.val
(3)E-> T	E.val=T.val
(4)T-> T <sub>1</sub> * F	T.val= T <sub>1</sub> .valxF.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}
$(4)T \rightarrow T_1 * F \{T.val = T_1.valxF.val\}$
(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.)

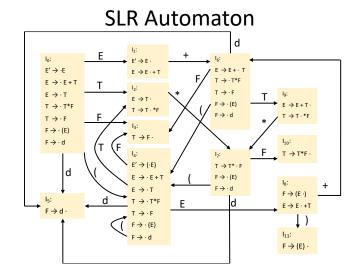


- If the underlying grammar of S-SDD is LR parsable
  - ◆ Then the SDT can be implemented during LR parsing
- Implement the converted SDT by reduction[借助归约实现]
  - ◆ Executing the action along with the reduction of LHS <- RHS

```
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.valxF.val\} 
(5)T -> F \{T.val = F.val\} 
(6)F -> (E) \{F.val = E.val\} 
(7)F -> digit \{F.val = digit.lexval\}
```

SDT





#### Extend LR Parse Stack[扩展分析栈]

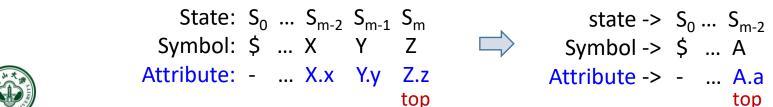


A.a

Y.v

X.x

- 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
  - 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
  - ◆ The manipulation can be done automatically by the parser

```
stack[top-2].val = f(stack[top-2].val, stack[top-1].val, stack[top].val)top = top-2

A.a

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

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

Y.y Z.Z top

state -> S<sub>0</sub> ... S<sub>m-2</sub> Symbol -> \$ ... A

Attribute -> - ... A.a

... A.a





#### 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 <sub>1</sub> + T	E.val= E <sub>1</sub> .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 <sub>1</sub> * F	T.val= T <sub>1</sub> .val*F.val	{stack[top-2].val * 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	



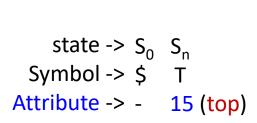


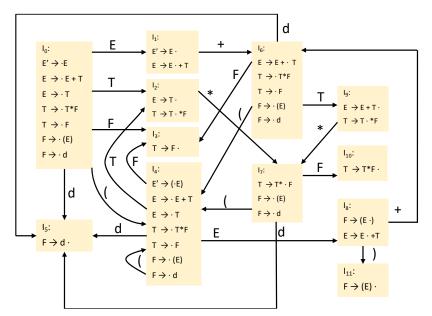
Productions	Semantic Actions
(1)L-> E	{ print(stack[top].val);}
(2)E-> E1+ T	{ stack[top-2].val=stack[top-2].val
(2)E > T	+ stack[top].val; top = top-2;}
(3)E-> T	
$(4)T->T_1*F$	{ stack[top-2].val=stack[top-2].val
	* stack[top].val; top = top-2;}
(5)T-> F	
(6)F-> (E)	{ stack[top-2].val= stack[top-1].val;
, , , ,	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)









n	n	H	t	•	2	*	5	+	/
	Μ	u	L	•	J		J		

#### **Productions**

(1)L-> E (2)E-> E1+ T

(3)E->T $(4)T->T_1*F$ 

(5)T-> F (6)F-> (E)

(7)F-> digit

Step	States (Illustrative)	Attributes	Input	Code	Output
1	\$	\$	3 * 5 + 4 n \$		
2	\$ 3	\$ 3	* 5 + 4 n \$		F  o digit
3	\$ F	\$ 3	* 5 + 4 n \$		$T \to F$
4	\$ T	\$ 3	* 5 + 4 n \$		
5	\$ T *	\$ 3 *	5 + 4 n \$		
6	\$T*5	\$ 3 * 5	+ 4 n \$		F  o digit
7	\$T*F	\$3*5	+ 4 n \$	3 * 5	T → T * F
8	\$ <u>T</u>	\$ 15	+ 4 n \$		E  o T
9	\$ E	\$ 15	+ 4 n \$		
10	\$ E +	\$ 15 +	4 n \$		
11	\$E+4	\$ 15 + 4	n \$		F  o digit
12	\$ E + F	\$ 15 + 4	n \$		$T \rightarrow F$
13	\$ E + T	\$ 15 + 4	n \$	15 + 4	$E \rightarrow E + T$
14	\$ E	\$ 19	n \$		
15	\$ E <b>n</b>	\$ 19 <b>n</b>	\$	print(19)	L → E <b>n</b>

\$

accept



\$ L

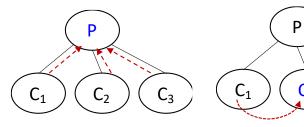
\$ 19

16

# **Semantic Analysis Revisit**



- SDD & SDT
- Synthesized & inherited attribute
  - Evaluation order DAG & topological order
- S-SDD & L-SDD
- S-SDD -> SDT -> Implementation
  - Postfix SDT
  - LR parser stack extension
  - Stack manipulation



Step	States (Illustrative)	Attributes	Input	Code	Output
1	\$	\$	3 * 5 + 4 n \$		
2	\$ 3	\$ 3	* 5 + 4 n \$		$F \rightarrow digit$
3	\$ F	\$ 3	* 5 + 4 n \$		$T \rightarrow F$
4	\$ T	\$ 3	* 5 + 4 n \$		
5	\$ T *	\$3*	5 + 4 n \$		
6	\$T*5	\$3 * 5	+ 4 n \$		F  o digit
7	\$T*F	\$3*5	+ 4 n \$	3 * 5	T → T * F
8	\$ T	\$ 15	+ 4 n \$		$E \rightarrow T$
9	\$ E	\$ 15	+ 4 n \$		
10	\$ E +	\$ 15 +	4 n \$		
11	\$E+4	\$ 15 + 4	n \$		$F \rightarrow digit$
12	\$ E + F	\$ 15 + 4	n \$		$T \rightarrow F$
13	\$ E + T	\$ 15 + 4	n \$	15 + 4	$E \rightarrow E + T$
14	\$ E	\$ 19	n \$		
15	\$ E <b>n</b>	\$ 19 <b>n</b>	\$	print(19)	$L \rightarrow E n$
16	\$ L	\$ 19	\$	accept	



# **Implement L-SDD**



- We have examined S-SDD -> SDT -> implementation
  - ◆ S-SDD can be converted to SDT with actions at production ends
  - ◆ The SDT can be parsed and translated by bottom-up, as long as the underlying grammar is LR-parsable
- What about the more-general L-attributed SDD?
  - ◆ Rule for turning L-SDD into an SDT
    - Embed the action that computes the inherited attributes for a nonterminal A immediately before the occurrence of A in the production body
    - 2. Place the actions that compute a synthesized attribute for the LHS at the end of the production body



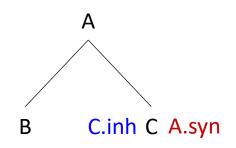


• A -> BC

◆C的继承属性:在C之前

◆A的综合属性: 在RHS末尾

Productions	Semantic rules
(1)T-> FT'	T'.inh = F.val
	T.val = T'.syn
(2)T'-> *FT <sub>1</sub> '	$T_1'$ .inh = T'.int * F.val
	$T'.syn = T_1'.syn$
(3)T'-> E	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 
$$\{T_1'.inh = T'.int * F.val\}\ T_1' \{T'.syn = T_1'.syn\}$$

(3) 
$$T' -> \mathcal{E} \{T'.syn = T'.inh\}$$



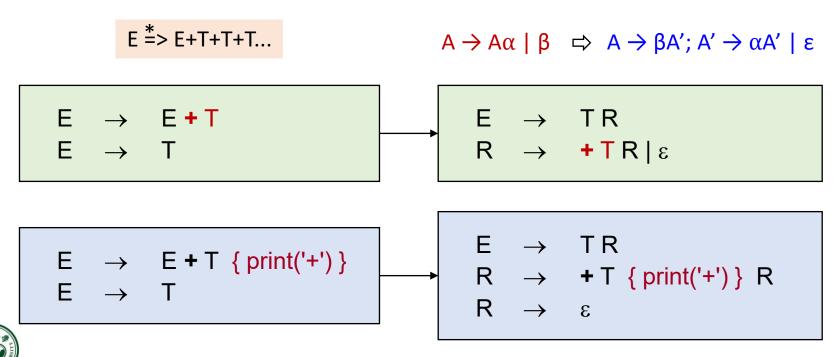
### **Implement L-SDD**



- If the underlying grammar is LL-parsable, then the SDT of L-SDD 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 parsing
  - ◆ An LL recursive-descent parser[递归下降的LL分析]
    - Augment non-terminal functions to both parsing and attributes handling
  - ◆ An LL predictive parser[非递归的LL预测分析]
    - Extend the parse stack to hold actions and certain data items needed for attribute evaluation
  - ◆ An LR parser[LR分析]
    - Rewrite grammars to involve marker
- For the top-down ones, we need to eliminate left recursion first!



- A motivating example: a simple case
  - Trick: treating actions as terminals if they do not calculate any attributes.





A more complex case

```
\begin{array}{ll} \mathsf{E} & \to \mathsf{E_1} + \mathsf{T} & \{ \, \mathsf{E.val} = \mathsf{E_1.val} + \mathsf{T.val}; \, \} \\ \mathsf{E} & \to \mathsf{E_1} - \mathsf{T} & \{ \, \mathsf{E.val} = \mathsf{E_1.val} - \mathsf{T.val}; \, \} \\ \mathsf{E} & \to \mathsf{T} & \{ \, \mathsf{E.val} = \mathsf{T.val}; \, \} \\ \mathsf{T} & \to \mathsf{(E)} & \{ \, \mathsf{T.val} = \mathsf{E.val}; \, \} \\ \mathsf{T} & \to \mathsf{num} & \{ \, \mathsf{T.val} = \mathsf{num.val}; \, \} \end{array}
```

```
A \rightarrow A\alpha \mid \beta
\downarrow \downarrow \downarrow
A \rightarrow \beta A';
A' \rightarrow \alpha A' \mid \epsilon
```





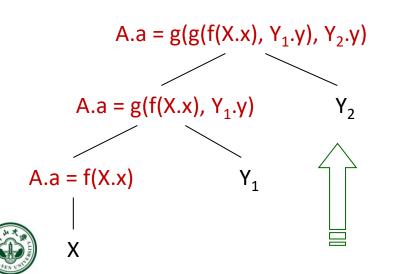
```
E \rightarrow T \{R.i = T.val;\} R \{E.val = R.s;\}
R \to +T \{R_1.i = R.i + T.val;\} R_1 \{R.s = R_1.s;\}
R \rightarrow -T \{R_1.i = R.i - T.val;\} R_1 \{R.s = R_1.s;\}
                                                                                 Input: 9 - 5 + 2
R \rightarrow \varepsilon \{ R.s = R.i; \}
T \rightarrow (E) \{ T.val = E.val; \}
                                                             E E.val = 6
T \rightarrow num \{ T.val = num.val; \}
                                       T.val = 9
                                    num.val = 9
                                                                          T.val = 5
                                                                                        ..... R.i = 4
                                                                                                  T.val = 2 ...... R.i = 6
                                                                       num.val = 5
                                                                                                num.val = 2
```



- General rules
  - Only available for S-SDD (postfix translation schemes), a subset of L-SDD

$$A \longrightarrow A_1 Y \{ A.a = g(A_1.a, Y.y) \}$$

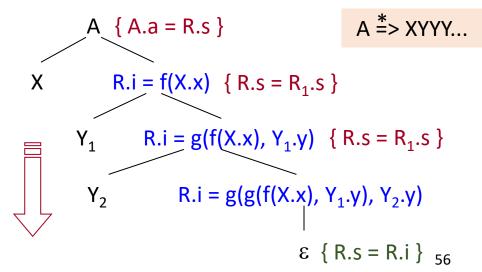
$$A \longrightarrow X \{ A.a = f(X.x) \}$$



```
A \rightarrow X { R.i = f(X.x) } R { A.a = R.s }

R \rightarrow Y { R<sub>1</sub>.i = g(R.i, Y.y) } R<sub>1</sub> { R.s = R<sub>1</sub>.s }

R \rightarrow \epsilon { R.s = R.i }
```



#### L-SDD in Recursive Decent Parsing



- A recursive-descent parser has a function A for each nonterminal A [递归预测分析方法]
  - Non-terminal expansion implemented by a function call
    - (Recursive) calls to functions for non-terminals in RHS
- Synthesized attributes: evaluate at end of function[综合属性: 最后计算]
  - ◆ All calls for RHS would have done by then
- Inherited attributes: pass as argument to function[继承属性:参数传递]
  - Values may come from parent or siblings
  - L-attributed guarantees they have been computed (can only come from already computed portion of RHS)

It becomes clearer here that the inherited attributes of symbols at RHS cannot depend on the synthesized attribute of the LHS



- Function arguments and return[参数和返回值]
  - Inherited: arguments
  - Synthesized: return

#### **Augmentation:**

- Use local variables[增加局部变量]
- Embed semantic actions[嵌入语义动作]

```
SDT

(1) T -> F \{T'.inh = F.val\} T' \{T.val = T'.syn\}
(2) T' -> *F \{T_1'.inh = T'.inh * F.val\} T_1' \{T'.syn = T_1'.syn\}
(3) T' -> \mathcal{E} \{T'.syn = T'.inh\}
(4) F -> digit \{F.val = digit.lexval\}
```

```
T'.syn T'(token, T'.inh) {
  if token = "*", then{
       getnext(token);
        F.val = F(token);
       T_1'.inh = T'.inh * F.val
       getnext(token);
       T_1'.syn = T_1'(token, T_1'.inh);
       T'.syn = T_1'.syn
        return T'.syn
   } else if token = "$", then {
       T'.syn = T'.inh;
        return T'.syn;
   } else
        Error;
```



#### L-SDD in LL Parsing[非递归预测]



- Extend the stack to hold actions 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 attributes for non-terminals
  - ◆ Typically, the data items are copies of attributes[属性备份]

Action	Code
А	Inh Attr.
A.syn	Syn Attr.

- Manage attributes on the stack[管理属性信息]
  - ◆ The inherited attributes of a nonterminal A are placed in the stack that represents that terminal[A的继承属性直接放在符号位]
    - 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[A的综合属性另存放在A后]

# 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 A.i, but on all attrs of B
  - ◆ As an L-SDD, it ensures A.i are available when A is on the stack top

    □ Thus, available to be copied into C
  - ◆ A's synthesized attrs A.s remain on the stack,
     below B and C when expansion happens

Action	Code
А	Inh Attr.
A.syn	Syn Attr.

Action	Code
В	Inh Attr.
B.syn	Syn Attr.
С	Inh Attr.
C.syn	Syn Attr.
A.syn	Syn Attr.



# L-SDD in LL Parsing (cont.)



- A-> BC: C.i may depend not only on the inherited attr A.I, 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 A.i will be gone, along with its stack record

Action Code

 左部展开时(i.e.,左部符号本身的记录出栈时),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

2. 综合属性出栈时, 要将综合属性值复制给后面的动作记录



Inh Attr.

Syn Attr.

Inh Attr.

Syn Attr.

Syn Attr.

B.syn

C.syn

A.syn



Three kinds of symbols:

- (1) Terminal
- (2) Non-terminal
- (3) Action symbol

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

(2) T'-> *F {T<sub>1</sub>'.inh = T'.int x F.val} T<sub>1</sub>' {T'.syn = T<sub>1</sub>'.syn}

(3) T'-> E {T'.syn = T'.inh}

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





### **Example (cont.)**



Input: 3\*5

Stack top 'digit' matches the input '3'

pop 'digit', but value copy is needed

$$(1)T-> F \{a_1\} \ T' \{a_2\} \qquad a_1:T'.inh = F.val \\ a_2:T.val = T'.syn \\ (2)T'-> * F \{a_3\} \ T_1' \{a_4\} \qquad a_3:T_1' inh = T'.inh \ x \ F.val \\ a_4:T'.syn = T1' .syn \\ (3)T'-> \mathcal{E} \{a_5\} \qquad a_5:T'.syn = T'.inh \\ (4)F-> digit \{a_6\} \qquad a_6:F.val = digit.lexval$$

a6: stack[top-1].val= stack[top].d\_lexval

digit	{a <sub>6</sub> }	F'syn	{a <sub>1</sub> }	T'	T'syn	{a <sub>2</sub> }	Tsyn	\$
lexv=3	d_lexv=3	val=3	val=3	inh	val		val	
	1		1					



### L-SDD in LR Parsing



- What we already learnt
  - ◆ LR > LL, w.r.t parsing power
    - We can do bottom-up if we can do top-down
- (1) T -> F {T'.inh = F.val} T' {T.val = T'.syn} (2) T'-> \*F {T<sub>1</sub>'.inh = T'.int x F.val} T<sub>1</sub>' {T'.syn = T<sub>1</sub>'.syn} (3) T'-> E {T'.syn = T'.inh } (4) F -> digit {F.val = digit.lexval }
- S-attributed SDD can be implemented in bottom-up way
  - All semantic actions are at the end of productions, i.e., triggered in reduce
- For L-SDD on an LL grammar, can it be implemented during bottom-up parsing?
  - Problem: semantic actions can be in anywhere of the production body



#### L-SDD in LR - The Problem



- It is not natural to evaluate inherited attributes
  - ◆ Example: how to get T'.inh
- Claims: inherited attributes are on the stack
  - ◆ L-SDD guarantee they've already been computed

◆ But computed by previous productions - deep in the stack

#### Solution

Hack the stack to dig out those values

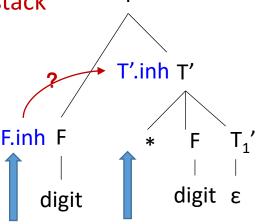
```
(1) T -> F {T'.inh = F.val} T' {T.val = T'.syn}

(2) T'-> *F {T<sub>1</sub>'.inh = T'.int x F.val} T<sub>1</sub>' {T'.syn = T<sub>1</sub>'.syn}

(3) T'-> E {T'.syn = T'.inh }

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





#### L-SDD in LR - Marker



- Given the following SDD, where  $|\alpha| \neq |\beta|$ 
  - $\bullet$  A-> X $\alpha$  {Y.in= X.s} Y | X $\beta$  {Y.in= X.s} Y
  - ♦ Y-> γ {Y.s= f(Y.in)}
- Problem: cannot generate stack location for Y.in
  - ◆ Because X.s is at different **relative locations** from Y in the stack
- Solution: insert markers M<sub>1</sub>, M<sub>2</sub> right before Y
  - $\bullet$  A-> X  $\alpha$   $M_1$ Y | X  $\beta$   $M_2$  Y
  - $M_1 \rightarrow \varepsilon$  { $M_1.s = stack[top-|\alpha|].s$ } // M1.s = X.s
  - $M_2$ ->  $\epsilon$  { $M_2$ .s= stack[top- $|\beta|$ ].s} // M2.s= X.s
  - Y -> γ {Y.s= f(stack[top-|γ|].s)} // Y.s= M1.s or Y.s= M2.s
- Marker: a non-terminal marking a location that is the same with the symbol that has an inherited attribute
  - Always produces ε since it is only a placeholder for an action

## **Modify Grammar with Marker**



- Given an L-SDD on an LL grammar, we can adapt the grammar to compute the same SDD during an LR parse
  - ◆ Put a marker non-terminal[标记非终结符] in the place of each embedded action

     Each of such places gets a distinct marker, say M, where M -> ε [空产生式]
  - Modify the action {a} if marker M replaces it in some production A-> $\alpha$  {a}  $\beta$ , and associate with M ->  $\epsilon$  an action {a'} that
    - Copies, as inherited attrs of M, any attrs of A or symbols of α that action {a} needs (e.g., M.i=A.i)[M的继承属性为{a}的输入]
    - 2. Compute sattrs in the same way as {a}, but makes those attrs be synthesized attrs of M (e.g., M.s= f(M.i)) [M的综合属性为{a}中的输出]

$$A \rightarrow \{B.i = f(A.i)\}$$
 BC

A-> 
$$M$$
 B C  
M->  $\epsilon$  {M.i=A.i; M.s=  $f(M.i)$ }





```
(1)T \rightarrow F \{T'.inh = F.val\} T' \{T.val = T'.syn\}

(2) T' \rightarrow F \{T'_1.inh = T'.int F.val\} T'_1 \{T'.syn = T'_1.syn\}

(3) T' \rightarrow \mathcal{E} \{T'.syn = T'.inh\}

(4) F \rightarrow digit \{F.val = digit.lexval\}
```



```
(1)T-> F M T' {T.val = T'.syn}

M->\varepsilon {M.i=F.val; M.s=M.i;}

(2) T'-> *F N T_1' {T'.syn = T_1'.syn}

N->\varepsilon {N.i1=T.inh; N.i2=F.val; N.s=M.i1 *M.i2 }

(3) T'-> \varepsilon {T'.syn = T'.inh }

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

$$A \rightarrow \{B.i = f(A.i)\}$$
 BC



$$M \rightarrow \varepsilon \{M.i=A.i; M.s=f(M.i)\}$$



### Stack Manipulation[栈操作]



```
(1)T \rightarrow F \{T'.inh = F.val\} T' \{T.val = T'.syn\}

(2) T' \rightarrow F \{T_1'.inh = T'.int \times F.val\} T_1' \{T'.syn = T_1'.syn\}

(3) T' \rightarrow \mathcal{E} \{T'.syn = T'.inh\}

(4) F \rightarrow digit \{F.val = digit.lexval\}
```

```
(1)T -> F M T' \{T.val = T'.syn\}

M -> \varepsilon \{M.i = F.val; M.s = M.i;\}

(2) T' -> *F N T_1' \{T'.syn = T_1'.syn\}

N -> \varepsilon \{N.i = T.inh; N.i = F.val; N.s = N.i =
```



```
(1)T-> FMT' {stack[top-2].val= stack[top].syn; top = top-2;} 

M->\epsilon {stack[top+1].T'inh= stack[top].val; top = top + 1; } 

(2)T'-> * FNT<sub>1</sub>' { stack[top-3].syn= stack[top].syn; top = top-3;} 

N->\epsilon {stack[top+1].T'inh= stack[top-2].T'inh * stack[top].val; top = top + 1; } 

(3)T'->\epsilon {stack[top+1].syn= stack[top].T'inh; top = top + 1;} 

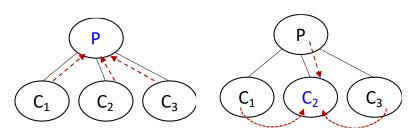
(4)F-> digit {stack[top].val=stack[top].lexval;}
```



### **Summary**



- SDD & SDT
- Synthesized & inherited attribute
  - Evaluation order DAG & topological order



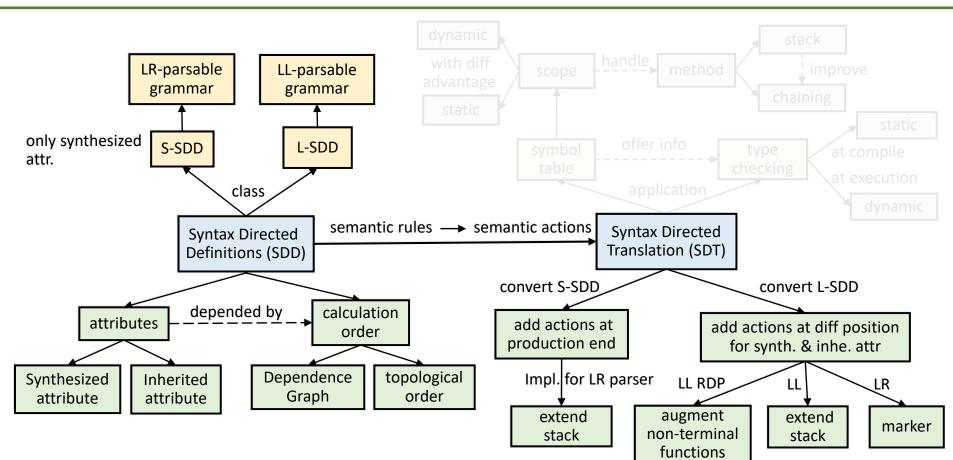
- S-SDD & L-SDD
- S-SDD -> SDT -> Implementation
  - (LR only) Postfix SDT -> stack extension -> stack manipulation
- L-SDD -> SDT -> Implementation
  - LL: Eliminate left recursion
    - RDP: inherited attributes (arguments) and synthesized attributes (return values)
    - Predictive Parsing: stack extension (action and synthesized records)



LR: Marker -> postfix SDT -> stack extension -> stack manipulation

#### Content





# **Further Reading**



- Dragon Book, 2<sup>nd</sup> Edition
  - ◆ Comprehensive Reading:
    - Section 5.1-5.3 on introduction to syntax-directed translation.
    - □ Section 5.4-5.5 on the implementation of translation schemes in top-down and bottom-up LR parsing.

