



编译原理

Compiler Principles

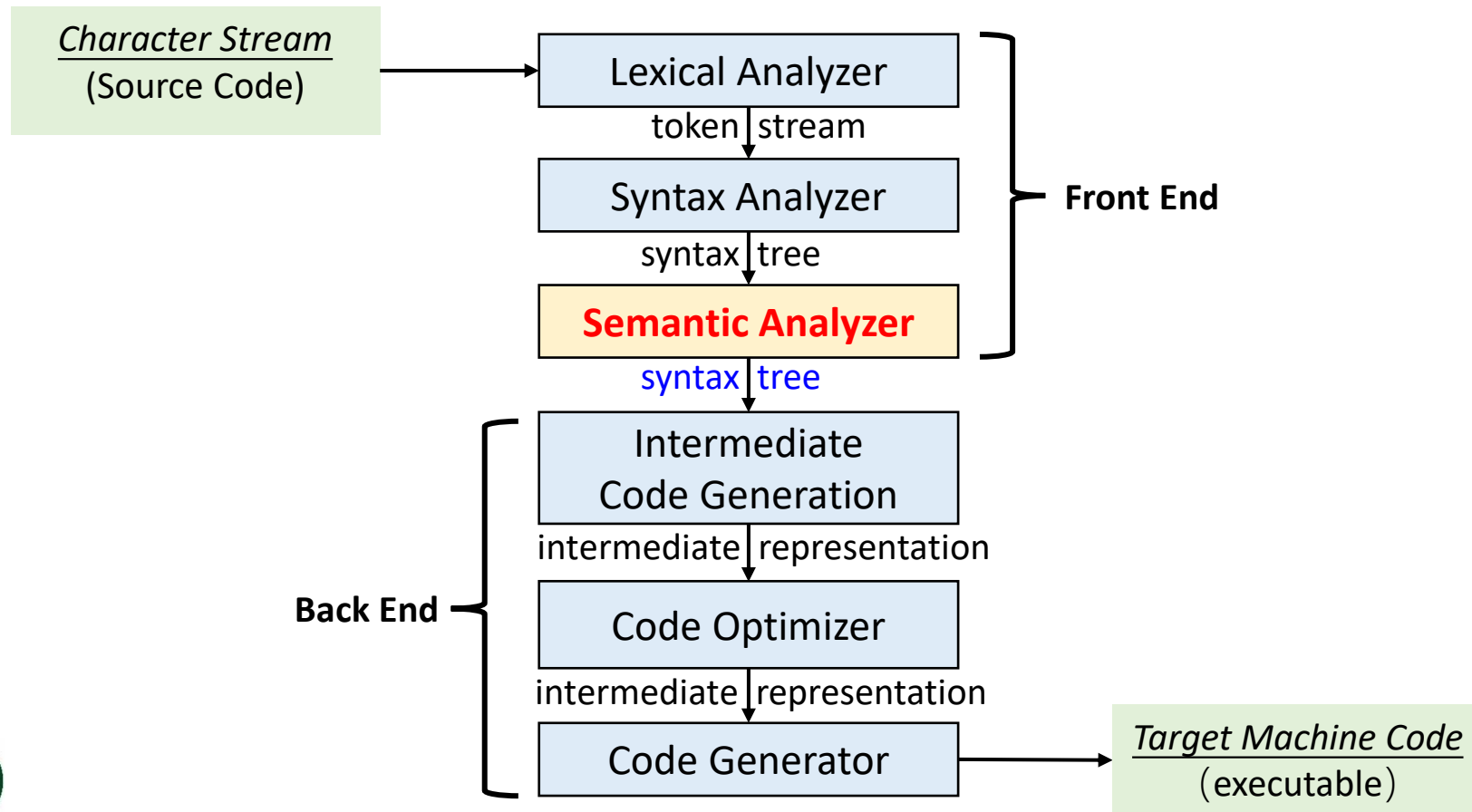
Lecture 6

Semantic Analysis: Intro & SDD & SDT

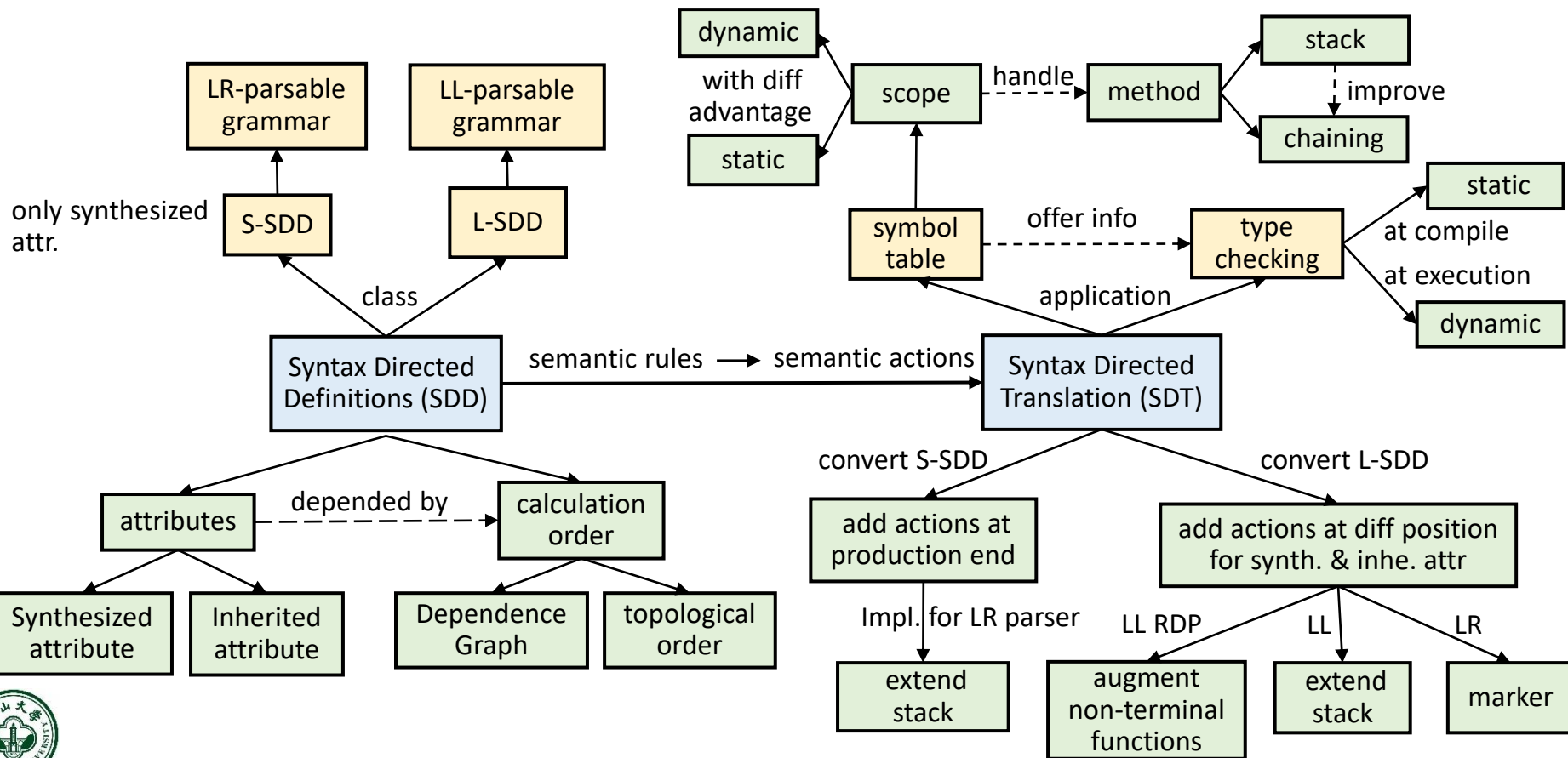
赵帅

计算机学院
中山大学

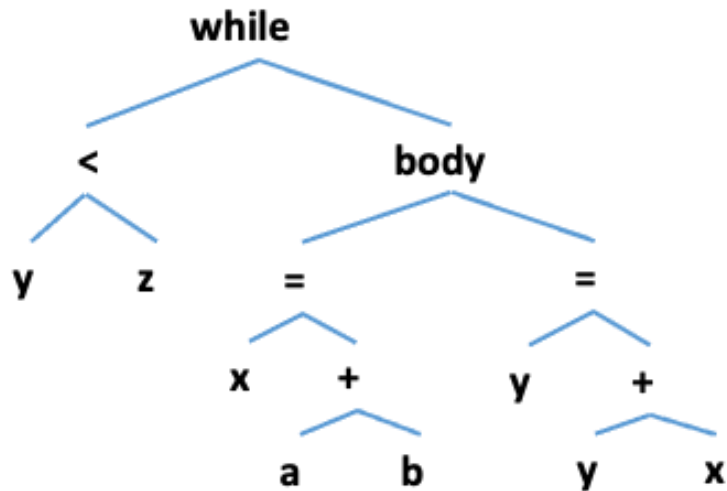
Compilation Phases[编译阶段]



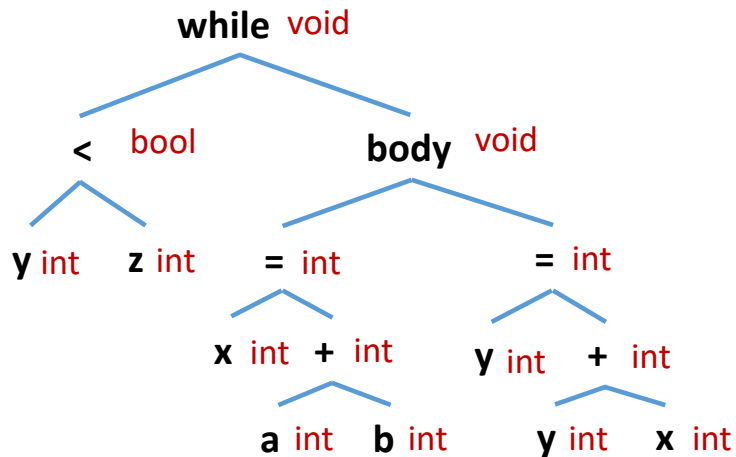
Content



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”
 - “it” --> “pizza”
 - ◆ 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]
 - ◆ Semantics: 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:
 $\{wcw \mid w \in (a|b)^*\}$
 - The first **w** represents the definition of an ID
 - The **c** represents arbitrary intervening code
 - The second **w** represents the use of the ID



- 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 last chance 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



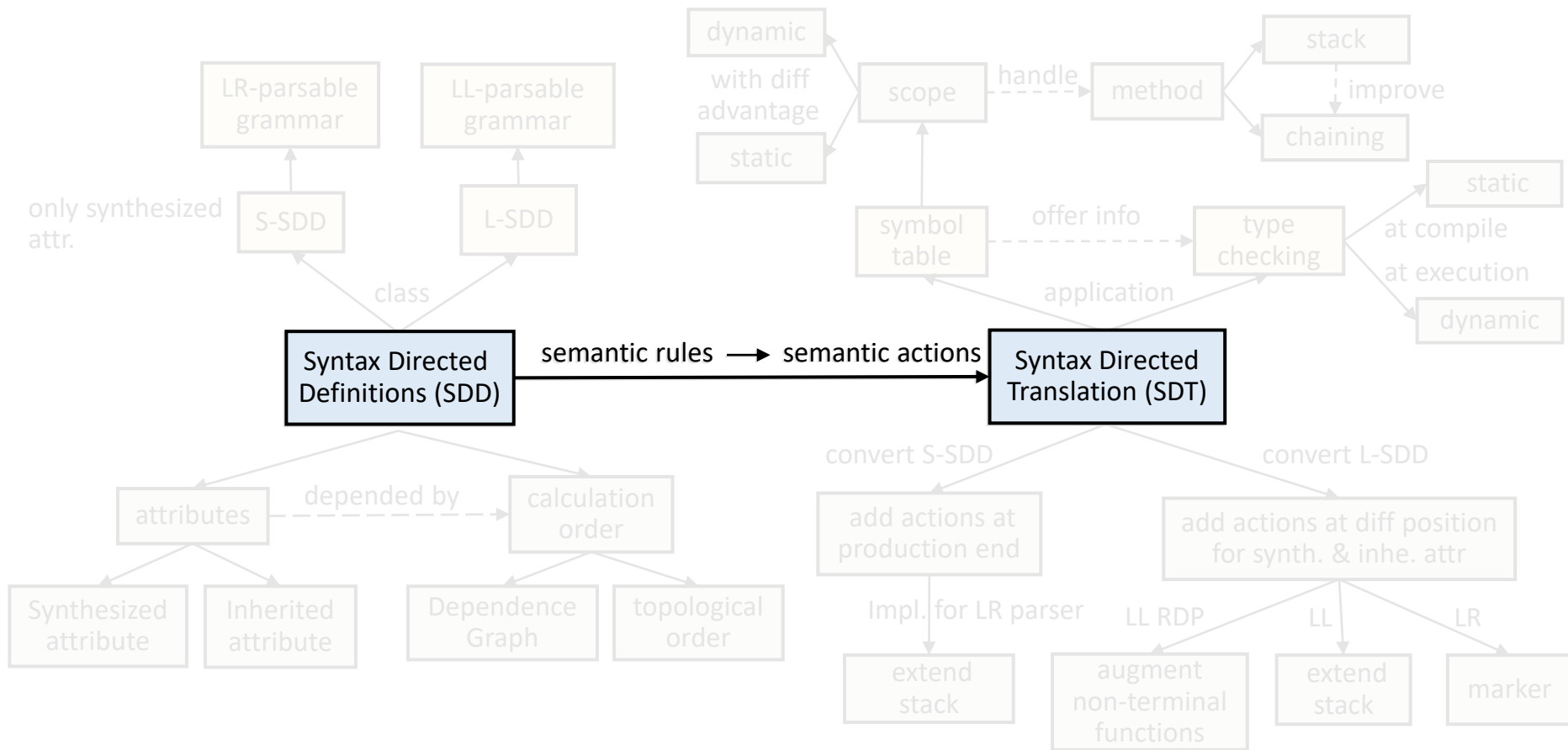
Semantic Analysis: Implementation



- 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



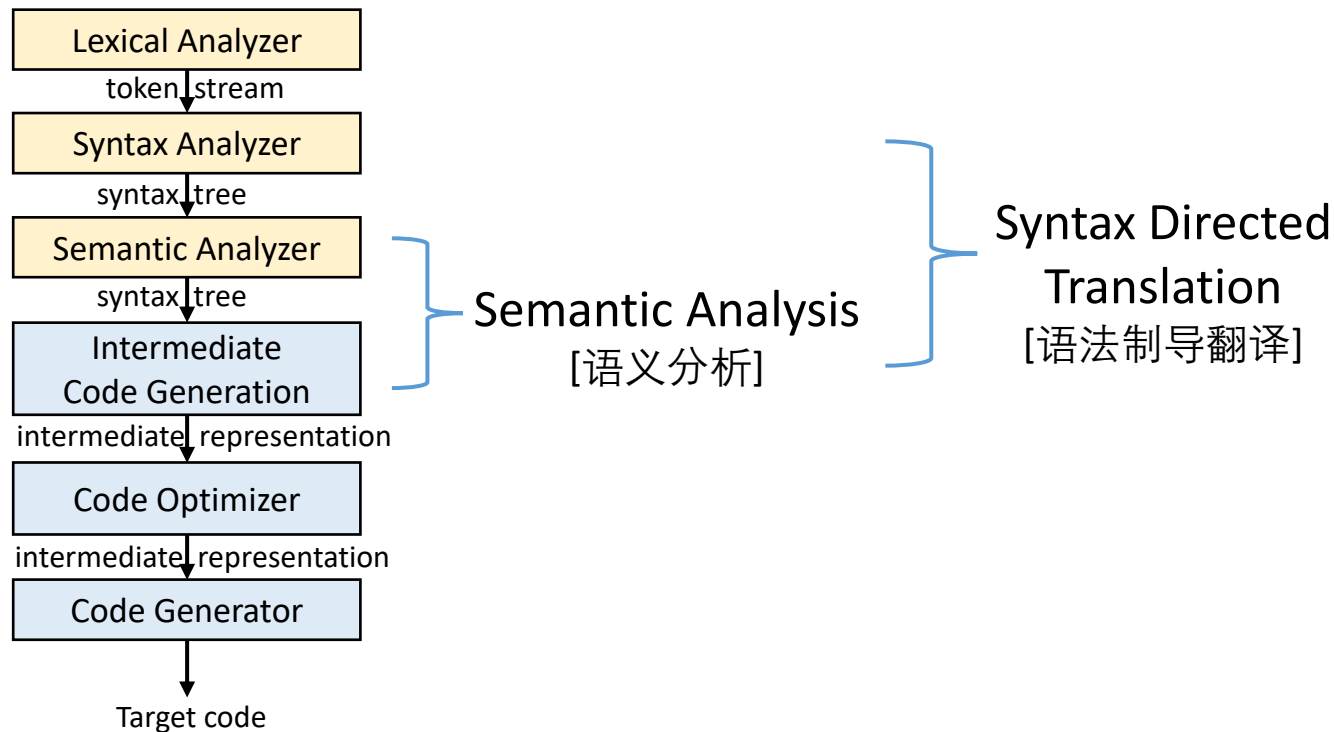
Content



Syntax Directed Translation



- **Syntax Directed Translation**[语法制导翻译]



- 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 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**)



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 |
|-------------------------|-------------------------|---------------------------|
| $E \rightarrow E1 + E2$ | $E.val; E1.val; E2.val$ | $E.val = E1.val + E2.val$ |
| $E \rightarrow id$ | $E.val; id.lexval$ | $E.val = id.lexval$ |



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

$E \rightarrow E1 + E2 \{E.val = E1.val + E2.val\}$

$E \rightarrow id \{E.val = id.lexval\}$

...

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 CFG grammar together with attributes and semantic rules
 - ◆ 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

Grammar

```
D -> TL
T -> int
T -> float
L -> L1, 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 -> {L1.inh = L.inh} L1, id
```

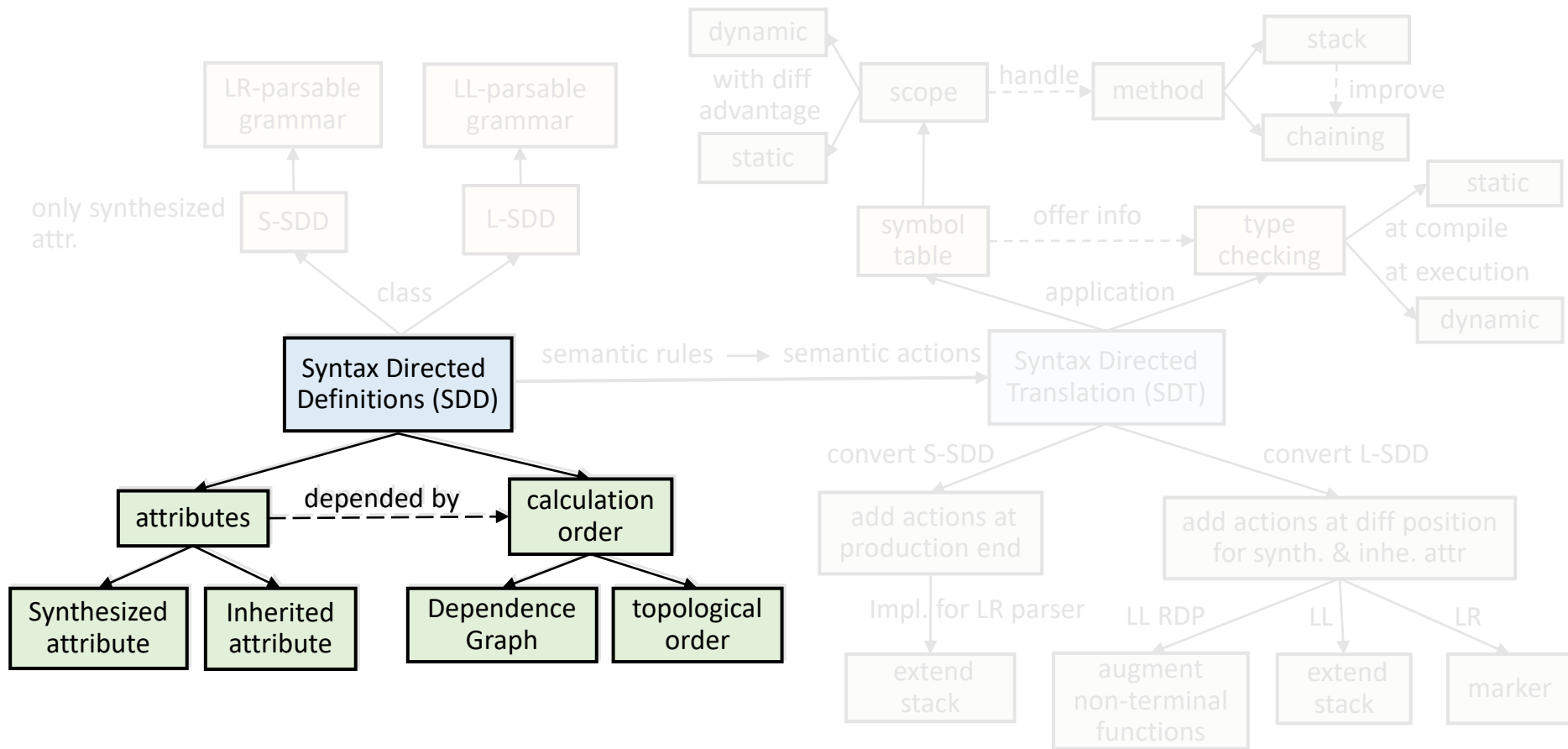


- Syntax: $A \rightarrow \alpha \{action_1\} \beta \{action_2\} \gamma \dots$
- Actions are executed “**at that point**” in the RHS
 - ◆ $action_1$ executes after α has been produced but before β
 - ◆ $action_2$ executes after α , $action_1$, β but before γ
- Semantic rule v.s. Action[语义规则 vs. 语义动作]
 - ◆ 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

$A \rightarrow \{ \dots \} X \{ \dots \} Y \{ \dots \}$


Semantic Actions

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 production at N
 - The production must have A as its head (i.e., $A \rightarrow \dots$)
 - ◆ 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., $\dots \rightarrow \dots A \dots$)
 - ◆ An inherited attribute 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]



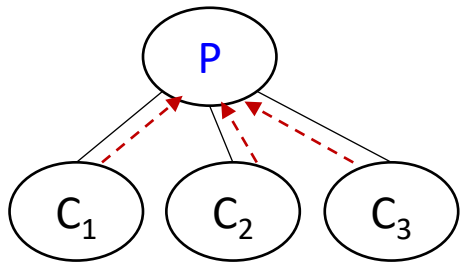
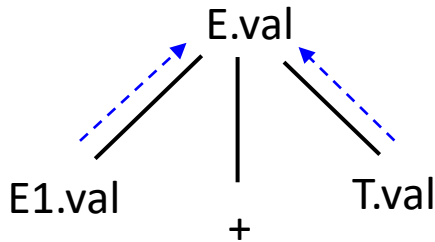
Synthesized Attribute[综合属性]



- 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
 - $P.\text{syn_attr} = f(P.\text{attrs}, C_1.\text{attrs}, C_2.\text{attrs}, C_3.\text{attrs})$
- ◆ Example

| Productions | Semantic rules |
|------------------------|---|
| $E \rightarrow E1 + T$ | $E.\text{val} = E1.\text{val} + T.\text{val}$ |



- 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]

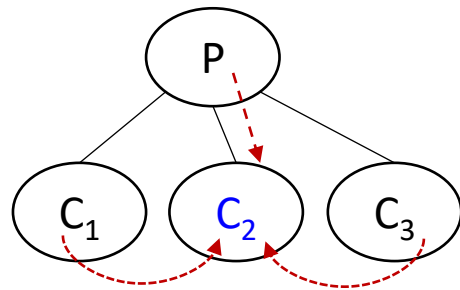
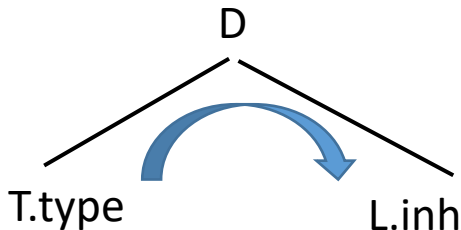


- Inherited attribute for non-terminal A of parse-tree node N[非终结符继承属性]

- ◆ 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 |
|---------------------|------------------|
| $D \rightarrow T L$ | $L.inh = T.type$ |

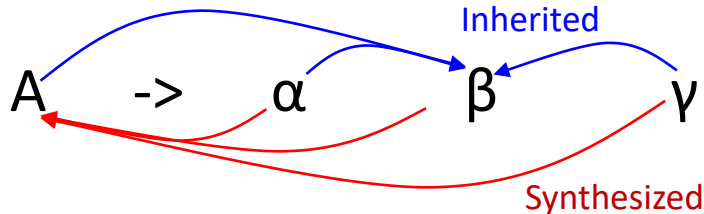


- Terminals **CANNOT** have inherited attributes[终结符无继承属性]

- ◆ Only synthesized attributes from lexical analysis



- 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
 - Why? **From A 's perspective α, β, γ are children**
- Or, b is an attribute in RHS (e.g., of β)
 - In which case b is an **inherited** attribute
 - Why? **From β 's perspective A, α, γ are parent or siblings**

Example: Synthesized attribute



Attribute grammar for simple integer arithmetic expression

| Grammar rule | Semantic Rules |
|------------------------------|----------------------------|
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| $E \rightarrow T$ | $E.val = T.val$ |
| $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| $T \rightarrow F$ | $T.val = F.val$ |
| $F \rightarrow (E)$ | $F.val = E.val$ |
| $F \rightarrow \text{dight}$ | $F.val = \text{dight.val}$ |

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



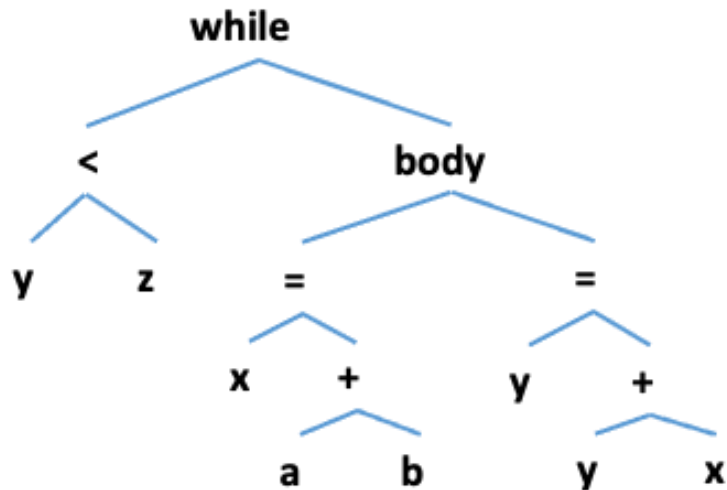
- Side effect[副作用]
 - ◆ 一般属性值计算（基于属性值或常量进行的）之外的功能
 - ◆ 例如：code generation, print results, modify symbol table ...
- Attribute grammar[属性文法]
 - ◆ 一个没有副作用的SDD
 - ◆ The rules define the value of an attribute purely in terms of the value of other attributes and constants[属性文法的规则仅仅通过其他属性值和常量来定义一个属性值]
- Annotated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



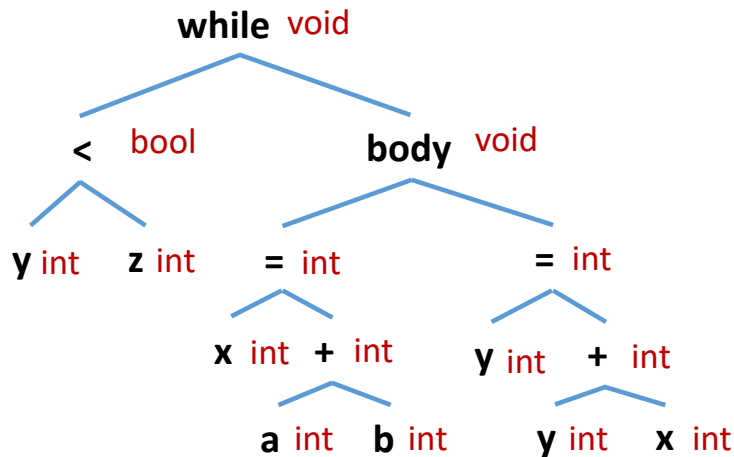
Compilation Phases[编译阶段]



A very simple illustrative case!



Abstract Syntax Tree(AST)



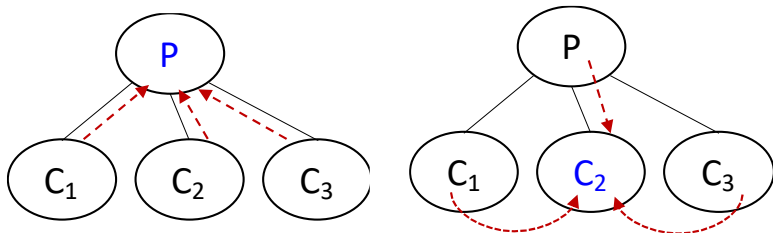
Annotated AST/Decorated AST
[带标注的抽象语法树]



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 \rightarrow TL
T \rightarrow int
T \rightarrow float
L \rightarrow L₁, id

SDD (rules)

L.inh = T.type
T.type = int
T.type = float
L₁.inh = L.inh

SDT (actions)

D \rightarrow T {L.inh = T.type} L
T \rightarrow int {T.type = int}
T \rightarrow float {T.type = float}
L \rightarrow {L₁.inh = L.inh} L₁, id

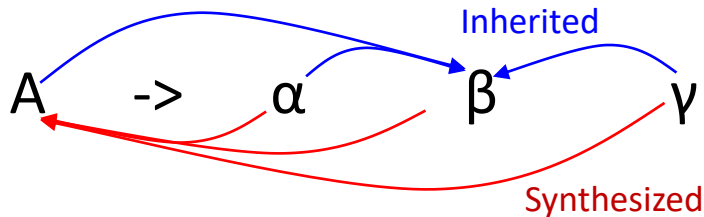


Dependence Graph[依赖图]



- 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**



- **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[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**



SDD:

Input:
float **a, b, c**



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 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, \dots, 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:

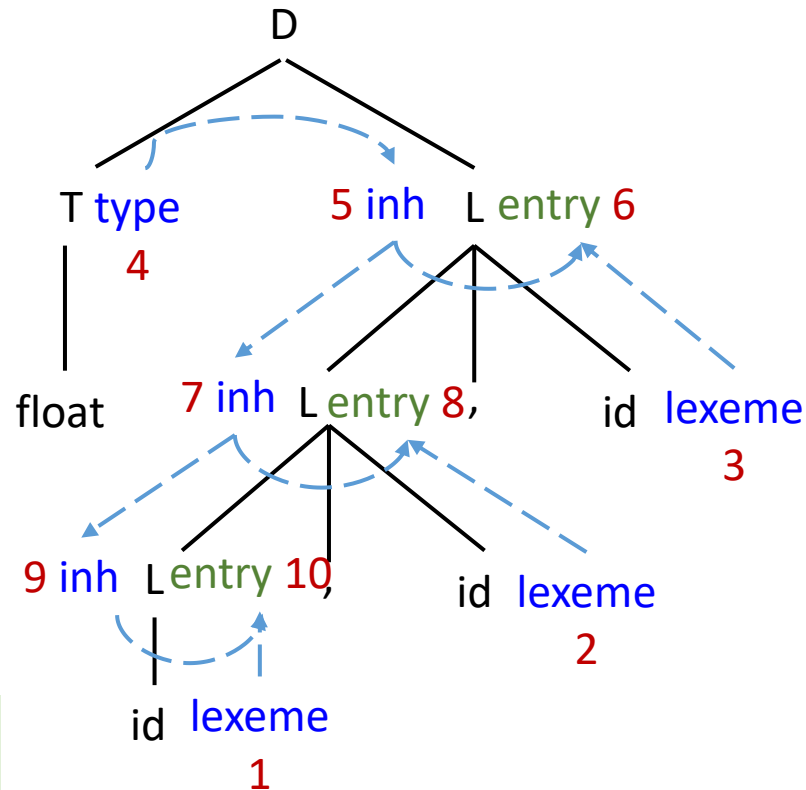
| Productions | Semantic rules |
|-------------------------|--|
| $D \rightarrow T L$ | $L.inh = T.type$ |
| $T \rightarrow int$ | $T.type = int$ |
| $T \rightarrow float$ | $T.type = float$ |
| $L \rightarrow L_1, id$ | $L1.inh = L.inh$ $addtype(id.entry, L.inh)$ |
| $L \rightarrow id$ | $addtype(id.entry, L.inh)$ |

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

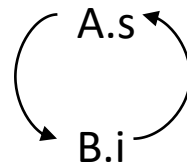
Production

$A \rightarrow B$

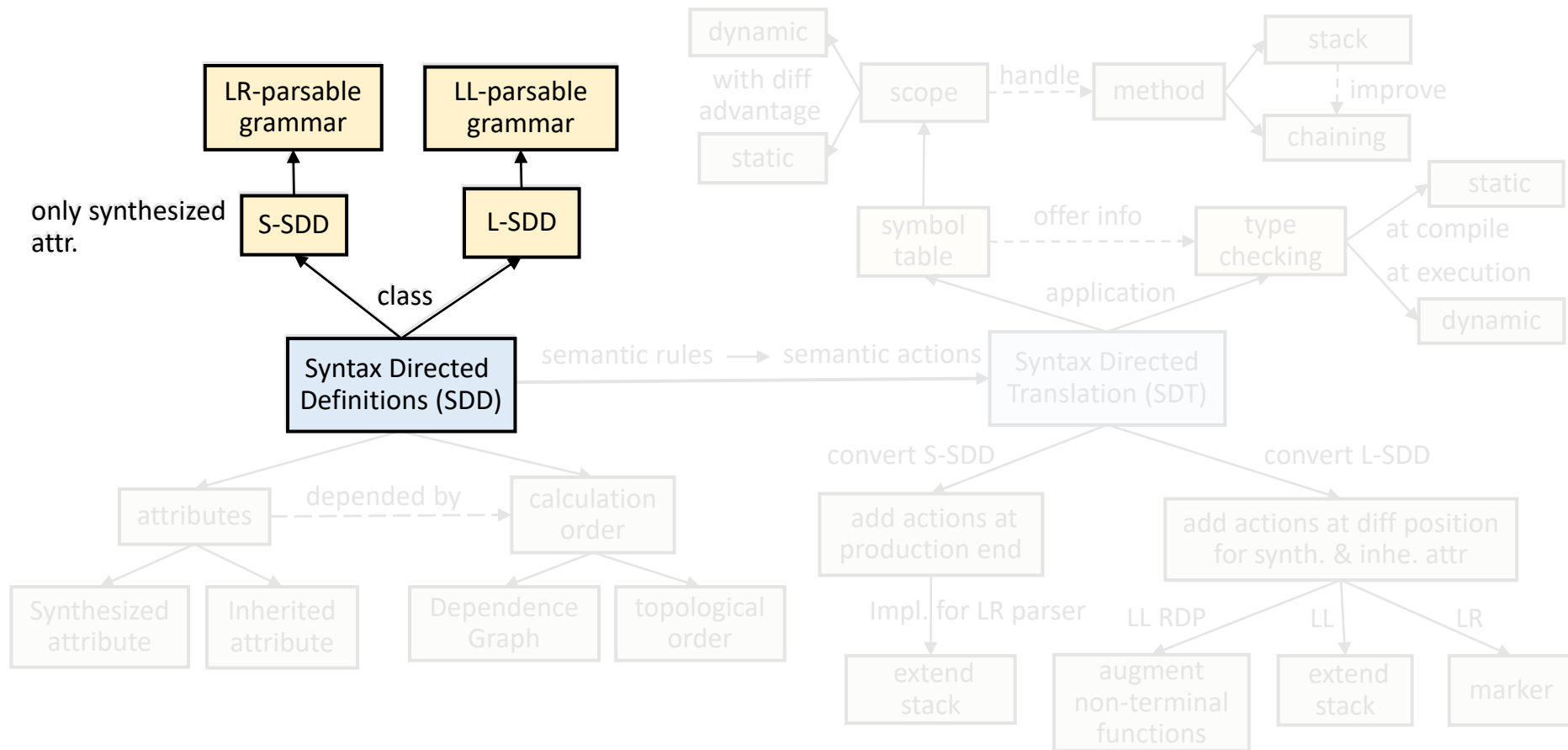
Semantic Rules

$A.s = B.i;$

$B.i = A.s + 1;$



Content



S-Attributed Definitions [S-属性定义]



- An SDD is **S-attributed** if **every attribute is synthesized** [只具有综合属性]
- If an SDD is S-attributed (S-SDD)
 - ◆ Can evaluate its attributes in **any bottom-up 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) $L \rightarrow E$ | <code>print(E.val)</code> |
| (2) $E \rightarrow E_1 + T$ | <code>E.val = E₁.val + T.val</code> |
| (3) $E \rightarrow T$ | <code>E.val = T.val</code> |
| (4) $T \rightarrow T_1 * F$ | <code>T.val = T₁.val * F.val</code> |
| (5) $T \rightarrow F$ | <code>T.val = F.val</code> |
| (6) $F \rightarrow (E)$ | <code>F.val = E.val</code> |
| (7) $F \rightarrow \text{digit}$ | <code>F.val = digit.lexval</code> |



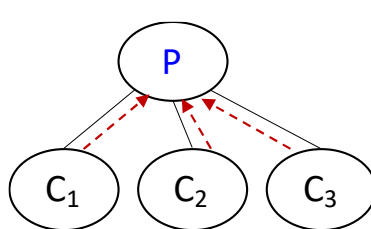
L-Attributed Definitions [L-属性定义]



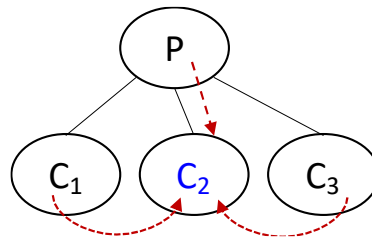
- An SDD is **L-attributed** (L-SDD) if
 - ◆ Between the attributes associated with a production body, **dependency graph edges can only go from left to right** [依赖图的边只能从左到右]
 - ◆ Each attribute can be either **synthesized, or inherited** with the rules that suppose $A \rightarrow X_1X_2...X_{i-1}X_i$, the inherited attribute $X_i.a$ only depends on:
 1. **Inherited** attributes associated with A [A的**继承**属性, Why?]
 2. Either **syn or inh** attributes of $X_1, X_2, ..., X_{i-1}$ located to the **left** of X_i [X_i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 **top-down/bottom-up parsing** [L属性的语义分析可在LL/LR分析中完成]



L-Attributed Definitions (cont.)



综合属性



继承属性

S-SDD or L-SDD?

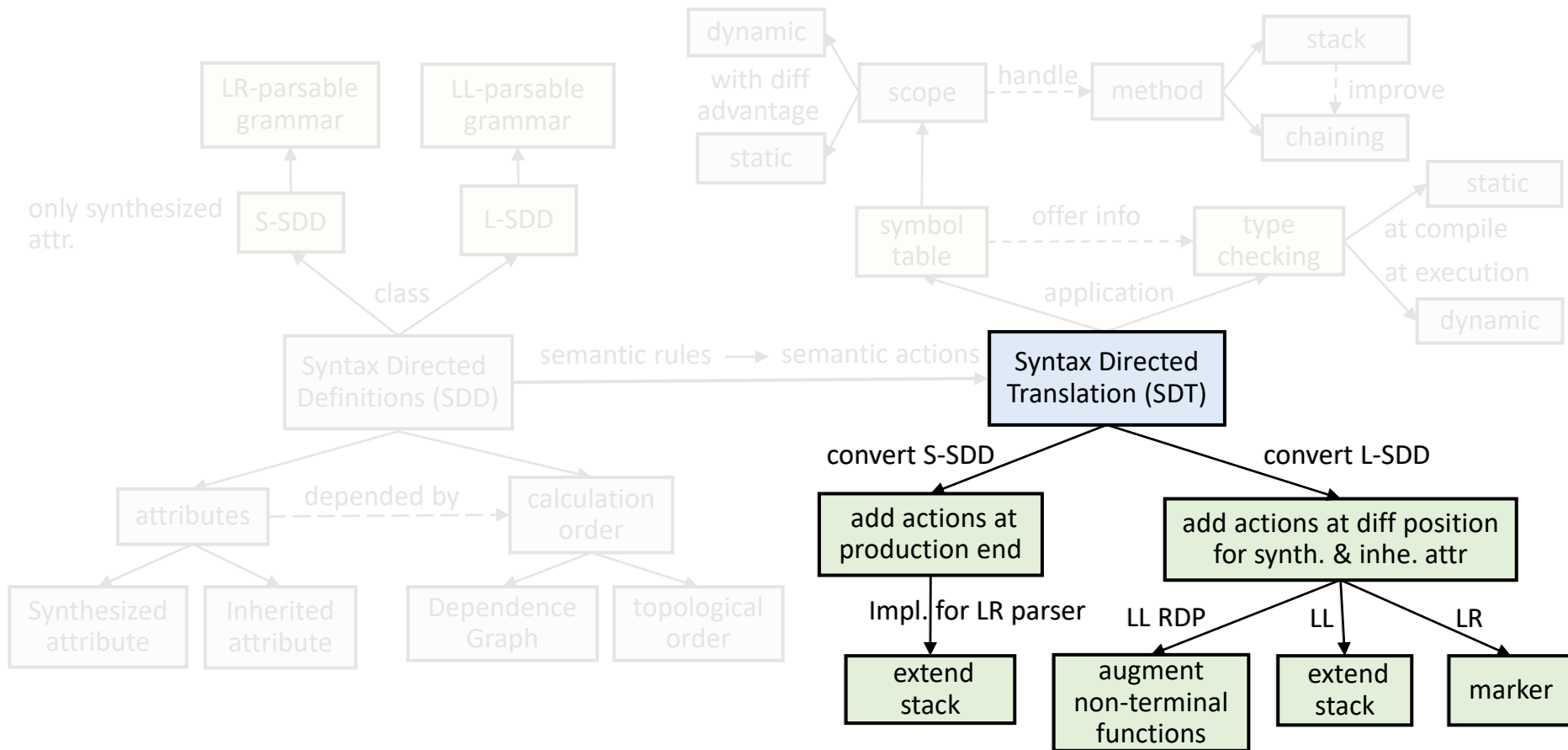
| Productions | Semantic rules |
|--------------------|------------------------------------|
| $A \rightarrow BC$ | $A.s = B.b$ $B.i = f(C.c, A.s)$ |

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 attr, 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



- 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
 - 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

- 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[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** [后綴/尾部SDT]

S-SDD

| Productions | Semantic rules |
|-----------------------------|---------------------------|
| (1) $L \rightarrow E$ | $print(E.val)$ |
| (2) $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| (3) $E \rightarrow T$ | $E.val = T.val$ |
| (4) $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| (5) $T \rightarrow F$ | $T.val = F.val$ |
| (6) $F \rightarrow (E)$ | $F.val = E.val$ |
| (7) $F \rightarrow digit$ | $F.val = digit.lexval$ |

SDT

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



Implement S-SDD (cont.)



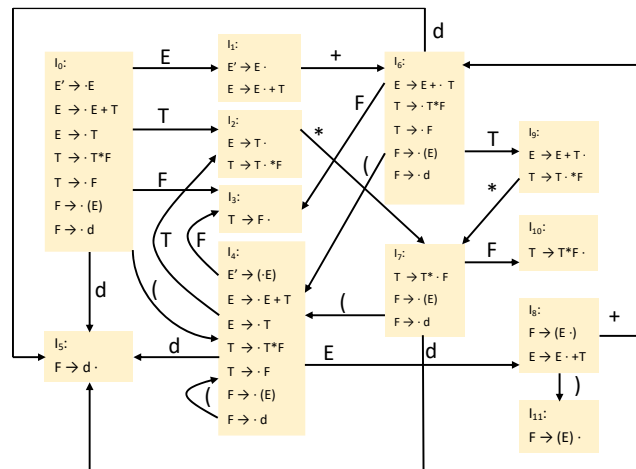
- If the underlying grammar of S-SDD is **LR parsable**
 - ◆ Then the SDT can be implemented during LR parsing
- Implement the converted SDT **by reduction** [借助归约实现]
 - ◆ Executing the action along with the reduction of LHS \leftarrow RHS

SDT

CFG with actions

- (1) $L \rightarrow E$ {*print(E.val)*}
- (2) $E \rightarrow E_1 + T$ {*E.val = E₁.val + T.val*}
- (3) $E \rightarrow T$ {*E.val = T.val*}
- (4) $T \rightarrow T_1 * F$ {*T.val = T₁.val * F.val*}
- (5) $T \rightarrow F$ {*T.val = F.val*}
- (6) $F \rightarrow (E)$ {*F.val = E.val*}
- (7) $F \rightarrow \text{digit}$ {*F.val = digit.lexval*}

SLR Automaton



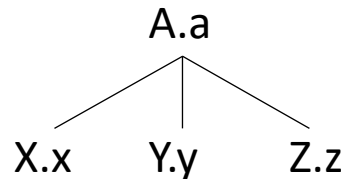
Extend LR Parse Stack[扩展分析栈]



- 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 \rightarrow XYZ**

- ◆ **x, y, z** are attributes of **X, Y, Z** respectively
- ◆ After the action, **A** and its attributes are at the top (i.e., $m-2$)



| | | | | | |
|------------|-------|-----|-----------|-----------|-------|
| State: | S_0 | ... | S_{m-2} | S_{m-1} | S_m |
| Symbol: | \$ | ... | X | Y | Z |
| Attribute: | - | ... | X.x | Y.y | Z.z |
| | | | | | top |



| | | | |
|-------------------------|-------|-----|-----------|
| state \rightarrow | S_0 | ... | S_{m-2} |
| Symbol \rightarrow | \$ | ... | A |
| Attribute \rightarrow | - | ... | A.a |
| | | | top |



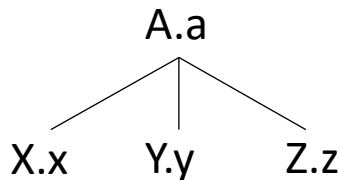
Stack Manipulation [栈操作]



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

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

$A \rightarrow XYZ \{A.a = f(X.x, Y.y, Z.z)\}$



State: $S_0 \dots S_{m-2} S_{m-1} S_m$
Symbol: $\$ \dots X \quad Y \quad Z$
Attribute: $- \dots X.x \quad Y.y \quad Z.z$
top



state $\rightarrow S_0 \dots S_{m-2}$
Symbol $\rightarrow \$ \dots A$
Attribute $\rightarrow - \dots A.a$
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) $L \rightarrow E$ | $\text{print}(E.\text{val})$ | $\{\text{print}(\text{stack}[\text{top}].\text{val});\}$ |
| (2) $E \rightarrow E_1 + T$ | $E.\text{val} = E_1.\text{val} + T.\text{val}$ | $\{\text{stack}[\text{top}-2].\text{val} = \text{stack}[\text{top}-2].\text{val} + \text{stack}[\text{top}].\text{val};$ $\text{top} = \text{top}-2;\}$ |
| (3) $E \rightarrow T$ | $E.\text{val} = T.\text{val}$ | |
| (4) $T \rightarrow T_1 * F$ | $T.\text{val} = T_1.\text{val} * F.\text{val}$ | $\{\text{stack}[\text{top}-2].\text{val} = \text{stack}[\text{top}-2].\text{val} * \text{stack}[\text{top}].\text{val};$ $\text{top} = \text{top}-2;\}$ |
| (5) $T \rightarrow F$ | $T.\text{val} = F.\text{val}$ | |
| (6) $F \rightarrow (E)$ | $F.\text{val} = E.\text{val}$ | $\{\text{stack}[\text{top}-2].\text{val} = (\text{stack}[\text{top}-1].\text{val});$ $\text{top} = \text{top}-2;\}$ |
| (7) $F \rightarrow \text{digit}$ | $F.\text{val} = \text{digit.lexval}$ | |

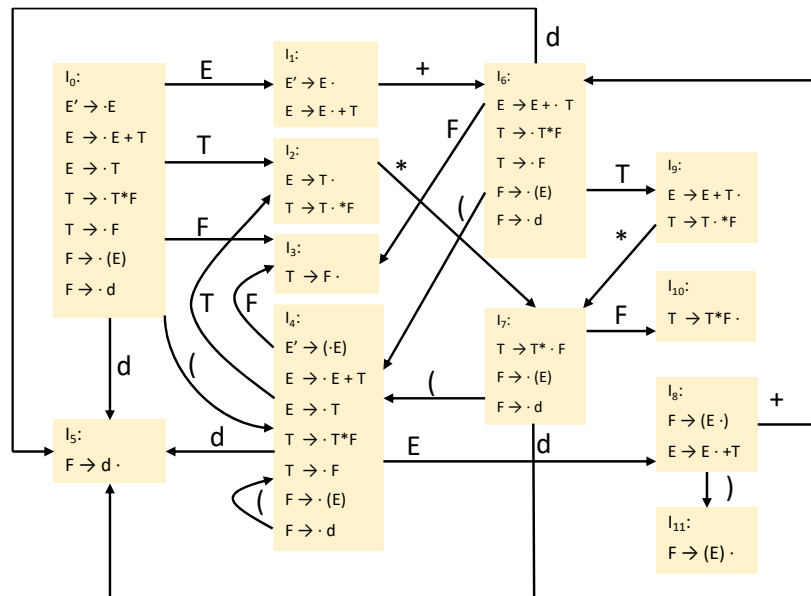


Example



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

Input: 3*5+4



state $\rightarrow S_0 S_2 S_7 S_{10}$
 Symbol $\rightarrow \$ T * F$
 Attribute $\rightarrow - 3 - 5 (\text{top})$



state $\rightarrow S_0 S_n$
 Symbol $\rightarrow \$ T$
 Attribute $\rightarrow - 15 (\text{top})$



Example



Input: $3*5+4$

Productions

- (1) $L \rightarrow E n$
- (2) $E \rightarrow E1 + T$
- (3) $E \rightarrow T$
- (4) $T \rightarrow T_1 * F$
- (5) $T \rightarrow F$
- (6) $F \rightarrow (E)$
- (7) $F \rightarrow \text{digit}$

| Step | States (Illustrative) | Attributes | Input | Code | Output |
|------|--------------------------|----------------|-----------------------|-----------|------------------------------|
| 1 | \$ | \$ | 3 * 5 + 4 n \$ | | |
| 2 | \$ 3 | \$ 3 | * 5 + 4 n \$ | -- | $F \rightarrow \text{digit}$ |
| 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 \$ | -- | $F \rightarrow \text{digit}$ |
| 7 | \$ T * F | \$ 3 * 5 | + 4 n \$ | 3 * 5 | $T \rightarrow T * F$ |
| 8 | \$ T | \$ 15 | + 4 n \$ | -- | $E \rightarrow T$ |
| 9 | \$ E | \$ 15 | + 4 n \$ | | |
| 10 | \$ E + | \$ 15 + | 4 n \$ | | |
| 11 | \$ E + 4 | \$ 15 + 4 | n \$ | -- | $F \rightarrow \text{digit}$ |
| 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 n | \$ 19 n | \$ | print(19) | $L \rightarrow E n$ |
| 16 | \$ L | \$ 19 | \$ | accept | |



Implement L-SDD



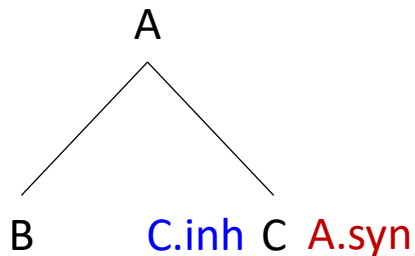
- We have examined **S-SDD** -> **SDT** -> **implementation**
 - ◆ S-SDD can be converted to SDT with **actions at production ends**
 - ◆ The SDT can be parsed and translated by **bottom-up**, as long as the underlying grammar is **LR-parsable**
- What about the more-general **L-attributed SDD**?
 - ◆ Rule for turning L-SDD into an SDT
 1. 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**



Example



- $A \rightarrow B \{C.inh\} C \{A.syn\}$
 - ◆ C的继承属性: 在C之前
 - ◆ A的综合属性: 在RHS末尾



| Productions | Semantic rules |
|-------------------------------|--|
| (1) $T \rightarrow FT'$ | $T'.inh = F.val$ $T.val = T'.syn$ |
| (2) $T' \rightarrow *FT_1'$ | $T_1'.inh = T'.int * F.val$ $T'.syn = T_1'.syn$ |
| (3) $T' \rightarrow \epsilon$ | $T'.syn = T'.inh$ |
| (4) $F \rightarrow digit$ | $F.val = digit.lexval$ |

SDT

- (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) $T' \rightarrow \epsilon \{T'.syn = T'.inh\}$
- (4) $F \rightarrow digit \{F.val = digit.lexval\}$



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!



Eliminating left recursion



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

$$E \stackrel{*}{\Rightarrow} E+T+T+T\dots$$

$$A \rightarrow A\alpha \mid \beta \quad \Leftrightarrow \quad A \rightarrow \beta A'; \quad A' \rightarrow \alpha A' \mid \varepsilon$$

$$\begin{array}{lcl} E & \rightarrow & E + T \\ E & \rightarrow & T \end{array}$$

$$\begin{array}{lcl} E & \rightarrow & T R \\ R & \rightarrow & + T R \mid \varepsilon \end{array}$$

$$\begin{array}{lcl} E & \rightarrow & E + T \quad \{ \text{print('+')} \} \\ E & \rightarrow & T \end{array}$$

a single terminal

$$\begin{array}{lcl} E & \rightarrow & T R \\ R & \rightarrow & + T \quad \{ \text{print('+')} \} R \\ R & \rightarrow & \varepsilon \end{array}$$



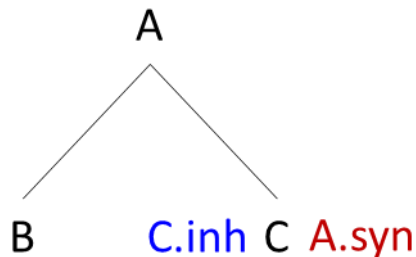
Semantic Analysis Revisit



- Implementing L-SDD: top-down/bottom-up
 - L-SDD \rightarrow SDT:
 1. 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**

$A \rightarrow B \{C.inh\} C \{A.syn\}$

- C的**继承**属性: **在C之前**
- A的**综合**属性: **在RHS末尾**



- Eliminating left recursion!



Eliminating left recursion



- A more complex case

```
E → E1 + T { E.val = E1.val + T.val; }  
E → E1 - T { E.val = E1.val - T.val; }  
E → T { E.val = T.val; }  
T → ( E ) { T.val = E.val; }  
T → num { T.val = num.val; }
```

```
E → T R  
R → + T R1  
R → - T R1  
R → ε  
T → ( E )  
T → num
```

$A \rightarrow A\alpha \mid \beta$



$A \rightarrow \beta A';$
 $A' \rightarrow \alpha A' \mid \epsilon$

```
E → T { R.i = T.val; } R { E.val = R.s; }  
R → + T { R1.i = R.i + T.val; } R1 { R.s = R1.s; }  
R → - T { R1.i = R.i - T.val; } R1 { R.s = R1.s; }  
R → ε { R.s = R.i; }  
T → ( E ) { T.val = E.val; }  
T → num { T.val = num.val; }
```

解决左递归斩断了产生式 (及actions)

$E \rightarrow E_1 + T \Rightarrow E \rightarrow TR \quad R \rightarrow +TR \quad R \rightarrow \epsilon$

用新引入的R, 传递属性值, 实现计算
R.i (属性传递) R.s (属性返回)

通过 $R \rightarrow \epsilon$, 将计算结果传回

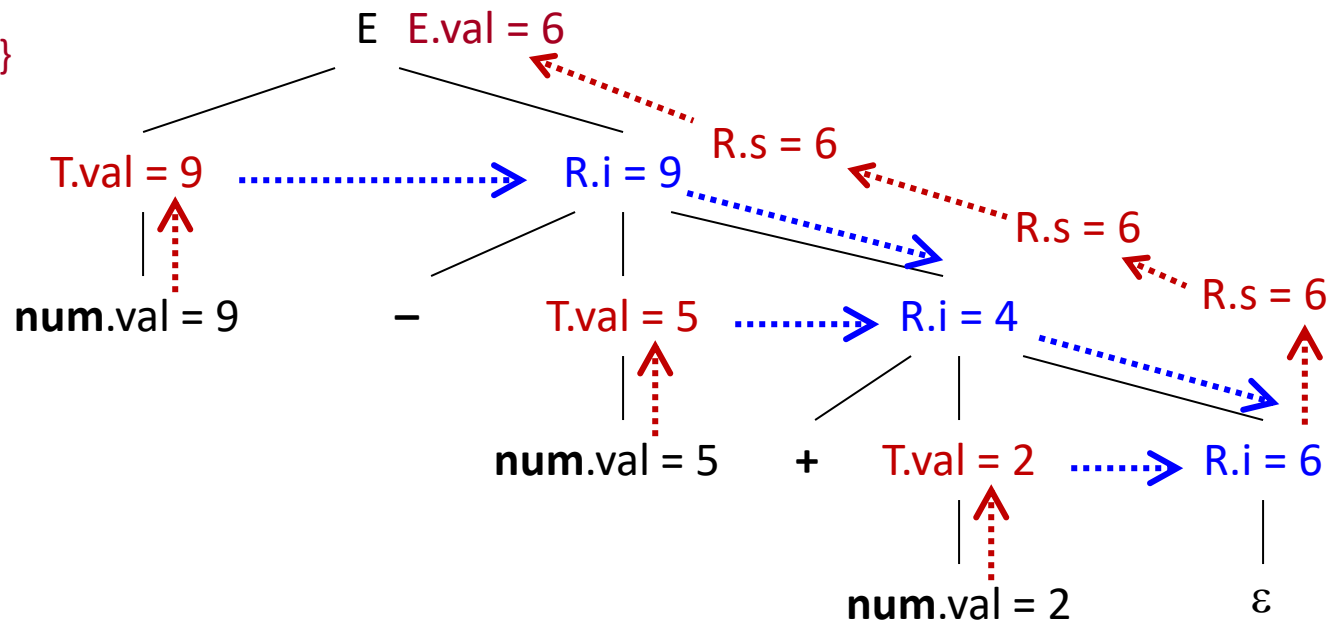


Eliminating left recursion



$E \rightarrow T \{ R.i = T.val; \} R \{ E.val = R.s; \}$
 $R \rightarrow + T \{ R_1.i = R.i + T.val; \} R_1 \{ R.s = R_1.s; \}$
 $R \rightarrow - T \{ R_1.i = R.i - T.val; \} R_1 \{ R.s = R_1.s; \}$
 $R \rightarrow \varepsilon \{ R.s = R.i; \}$
 $T \rightarrow (E) \{ T.val = E.val; \}$
 $T \rightarrow \text{num} \{ T.val = \text{num.val}; \}$

Input: 9 - 5 + 2



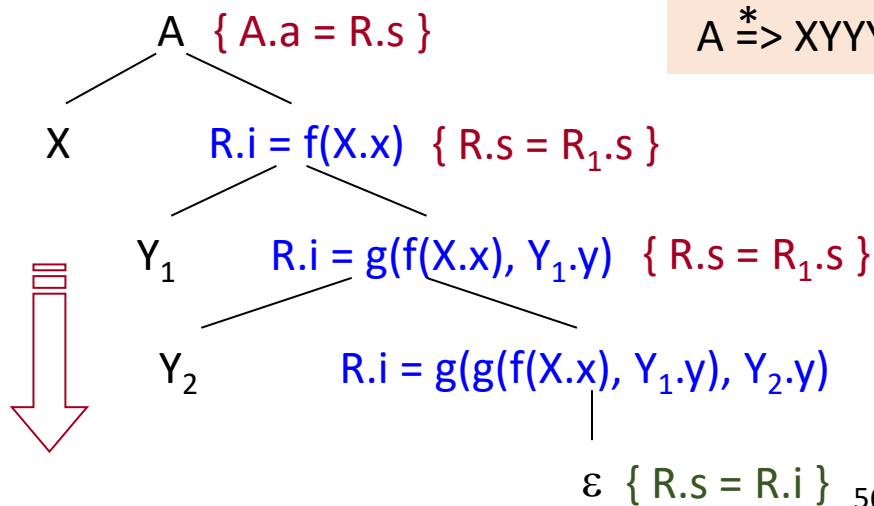
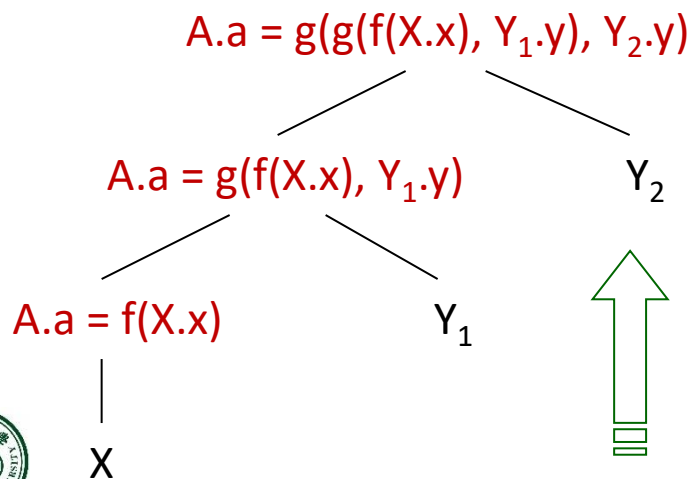
Eliminating left recursion



- General rules

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

$$\begin{aligned} A &\rightarrow A_1 Y \{ A.a = g(A_1.a, Y.y) \} \\ A &\rightarrow X \{ A.a = f(X.x) \} \end{aligned}$$

$$\begin{aligned} A &\rightarrow X \{ R.i = f(X.x) \} R \{ A.a = R.s \} \\ R &\rightarrow Y \{ R_1.i = g(R.i, Y.y) \} R_1 \{ R.s = R_1.s \} \\ R &\rightarrow \varepsilon \{ R.s = R.i \} \end{aligned}$$


$A \xRightarrow{*} XYYY\dots$



Semantic Analysis Revisit



- Evaluation order -> DAG -> topological sort -> circle is NOT allowed
- SDD Subsets without circle: **S-SDD** and **L-SDD**
 - **S-SDD**: **Synthesized** attributes only
 - **L-SDD**: **Synthesized** and **Inherited** attributes ($A \rightarrow X_1 X_2 \dots X_{i-1} X_i$)
 - $X_i.inh$ can depend on: (i) $A.inh$, (ii) $X_j.attr$, $j < i$, (iii) $X_i.attr$ with **no circles**
- Implementation:
 - Based on **parser tree** or **AST**
 - **Any computation order** without dependency circles
 - Along with Parser
 - Strictly follows the parsing order: Top-down or Bottom-up
L-SDD S/L-SDD



Semantic Analysis Revisit



- Implementing S-SDD: bottom-up
 - S-SDD \rightarrow SDT: Place each action at the end of the production
 - Slack manuscript: enable automation

S-SDD

| Productions | Semantic rules |
|-----------------------------|---------------------------|
| (1) $L \rightarrow E$ | $print(E.val)$ |
| (2) $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ |
| (3) $E \rightarrow T$ | $E.val = T.val$ |
| (4) $T \rightarrow T_1 * F$ | $T.val = T_1.val * F.val$ |
| (5) $T \rightarrow F$ | $T.val = F.val$ |
| (6) $F \rightarrow (E)$ | $F.val = E.val$ |
| (7) $F \rightarrow digit$ | $F.val = digit.lexval$ |

SDT

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

- Example: $A \rightarrow XYZ$

- x, y, z are attributes of X, Y, Z respectively
- After the action, A and its attributes are at the top (i.e., $m-2$)

State: $S_0 \dots S_{m-2} S_{m-1} S_m$
 Symbol: $\$ \dots X \quad Y \quad Z$
 Attribute: $- \dots X.x \quad Y.y \quad Z.z$
top



state $\rightarrow S_0 \dots S_{m-2}$
 Symbol $\rightarrow \$ \dots A$
 Attribute $\rightarrow - \dots A.a$
top



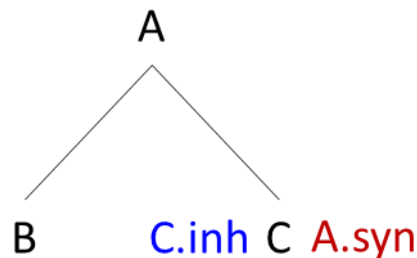
Semantic Analysis Revisit



- Implementing L-SDD: top-down/bottom-up
 - L-SDD \rightarrow SDT:
 1. 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**

$A \rightarrow B \{C.inh\} C \{A.syn\}$

- C的**继承**属性: **在C之前**
- A的**综合**属性: **在RHS末尾**



- Eliminating left recursion!



Eliminating left recursion



- A more complex case

```

E → E1 + T { E.val = E1.val + T.val; }
E → E1 - T { E.val = E1.val - T.val; }
E → T { E.val = T.val; }
T → ( E ) { T.val = E.val; }
T → num { T.val = num.val; }
    
```

```

E → T R
R → + T R1
R → - T R1
R → ε
T → ( E )
T → num
    
```

$A \rightarrow A\alpha \mid \beta$



$A \rightarrow \beta A';$
 $A' \rightarrow \alpha A' \mid \epsilon$

```

E → T { R.i = T.val; } R { E.val = R.s; }
R → + T { R1.i = R.i + T.val; } R1 { R.s = R1.s; }
R → - T { R1.i = R.i - T.val; } R1 { R.s = R1.s; }
R → ε { R.s = R.i; }
T → ( E ) { T.val = E.val; }
T → num { T.val = num.val; }
    
```

解决左递归斩断了产生式 (及actions)

$E \rightarrow E_1 + T \Rightarrow E \rightarrow TR \quad R \rightarrow +TR \quad R \rightarrow \epsilon$

用新引入的R, 传递属性值, 实现计算
 R.i (属性传递) R.s (属性返回)

通过R → ε, 将计算结果传回



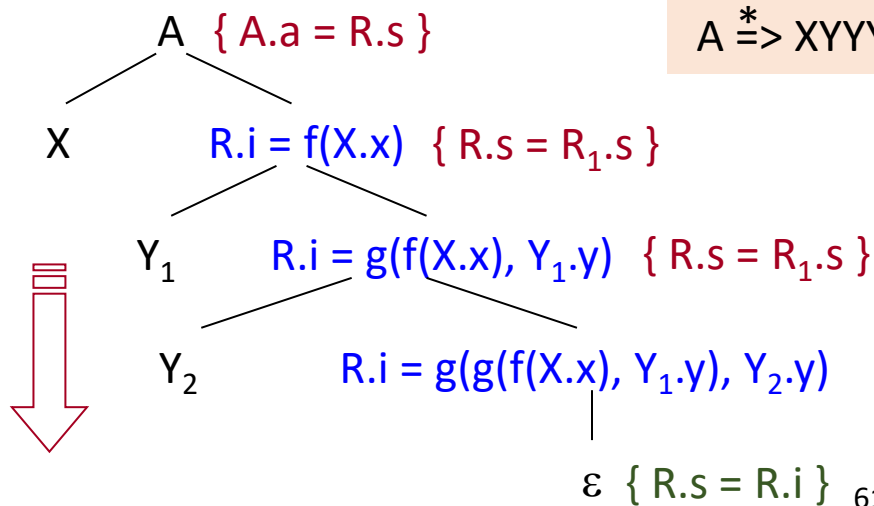
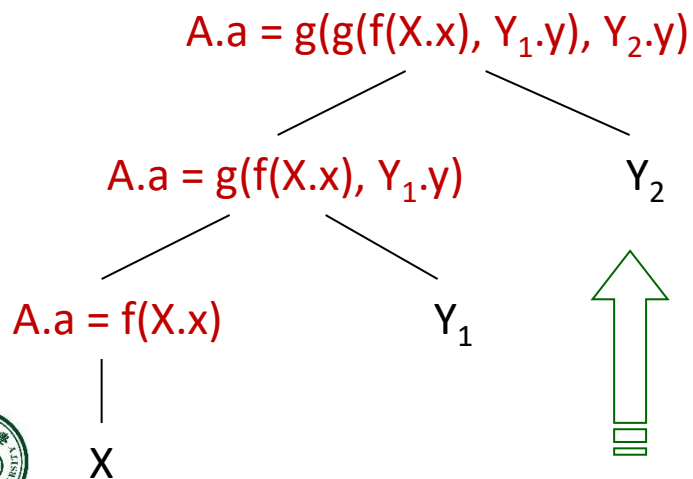
Eliminating left recursion



- General rules

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

$$\begin{aligned} A &\rightarrow A_1 Y \{ A.a = g(A_1.a, Y.y) \} \\ A &\rightarrow X \{ A.a = f(X.x) \} \end{aligned}$$

$$\begin{aligned} A &\rightarrow X \{ R.i = f(X.x) \} R \{ A.a = R.s \} \\ R &\rightarrow Y \{ R_1.i = g(R.i, Y.y) \} R_1 \{ R.s = R_1.s \} \\ R &\rightarrow \varepsilon \{ R.s = R.i \} \end{aligned}$$


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**



Example



- 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) $T \rightarrow F$ { $T'.inh = F.val$ } T' { $T.val = T'.syn$ }
- (2) $T' \rightarrow *F$ { $T_1'.inh = T'.inh * F.val$ } T_1' { $T'.syn = T_1'.syn$ }
- (3) $T' \rightarrow \epsilon$ { $T'.syn = T'.inh$ }
- (4) $F \rightarrow digit$ { $F.val = digit.lexval$ }

```
T'.syn T'(token, T'.inh) {  
  if token = "*", then {  
    getNext(token);  
    F.val = F(token);  
    T1'.inh = T'.inh * F.val;  
    getNext(token);  
    T1'.syn = T1'(token, T1'.inh);  
    T'.syn = T1'.syn;  
    return T'.syn;  
  } else if token = "$", then {  
    T'.syn = T'.inh;  
    return T'.syn;  
  } else  
    Error;  
}
```

D: **F.val** -> **T₁'.inh** -> **T₁'.syn** 63



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]
- 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]

| 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

- $A \rightarrow BC$, 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 |
|--------|-----------|
| A | Inh Attr. |
| A.syn | Syn Attr. |

| Action | Code |
|--------|-----------|
| B | Inh Attr. |
| B.syn | Syn Attr. |
| C | 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 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

1. 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

| Action | Code |
|--------|-----------|
| B | Inh Attr. |
| B.syn | Syn Attr. |
| C | Inh Attr. |
| C.syn | Syn Attr. |
| A.syn | Syn Attr. |



Example



Three kinds of symbols:

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

(1) $T \rightarrow F \{T'.inh = F.val\} \quad T' \{T.val = T'.syn\}$
(2) $T' \rightarrow *F \{T_1'.inh = T'.int \times F.val\} \quad T_1' \{T'.syn = T_1'.syn\}$
(3) $T' \rightarrow \varepsilon \{T'.syn = T'.inh\}$
(4) $F \rightarrow \text{digit} \{F.val = \text{digit.lexval}\}$



(1) $T \rightarrow F \{a_1\} \quad T' \{a_2\} \quad a_1: T'.inh = F.val$
 $a_2: T.val = T'.syn$
(2) $T' \rightarrow *F \{a_3\} \quad T_1' \{a_4\} \quad a_3: T_1'.inh = T'.inh \times F.val$
 $a_4: T'.syn = T_1'.syn$
(3) $T' \rightarrow \varepsilon \{a_5\} \quad a_5: T'.syn = T'.inh$
(4) $F \rightarrow \text{digit} \{a_6\} \quad a_6: F.val = \text{digit.lexval}$



Example (cont.)



Input: 3*5

Stack top 'digit' **matches** the input '3'

- pop 'digit', but value copy is needed

(1) $T \rightarrow F \{a_1\} T' \{a_2\}$ $a_1:T'.inh = F.val$
 $a_2:T.val = T'.syn$
(2) $T' \rightarrow * F \{a_3\} T_1' \{a_4\}$ $a_3:T_1'.inh = T'.inh \times F.val$
 $a_4:T'.syn = T_1'.syn$
(3) $T' \rightarrow \epsilon \{a_5\}$ $a_5:T'.syn = T'.inh$
(4) $F \rightarrow digit \{a_6\}$ $a_6:F.val = digit.lexval$

$T \rightarrow F \{a_1\} T' \{a_2\}$

$a_6: stack[top-1].val = stack[top].d_lexval$

$F \rightarrow digit \{a_6\}$

| digit | {a ₆ } | F'syn | {a ₁ } | T' | T'syn | {a ₂ } | Tsyn | \$ |
|--------|-------------------|-------|-------------------|-----|-------|-------------------|------|----|
| lexv=3 | d_lexv=3 | val=3 | val=3 | inh | val | | val | |

Diagram showing value copying: an arrow from the 'd_lexv=3' cell to the 'val=3' cell under 'F'syn', and another arrow from the 'val=3' cell under '{a₁}' to the 'val' cell under 'T'syn'.



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

- ◆ 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

(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) $T' \rightarrow \epsilon \{T'.syn = T'.inh\}$
(4) $F \rightarrow \text{digit} \{F.val = \text{digit.lexval}\}$

| Action | Code |
|--------|-----------|
| B | Inh Attr. |
| B.syn | Syn Attr. |
| C | Inh Attr. |
| C.syn | Syn Attr. |
| A.syn | Syn Attr. |



L-SDD in LR - The Problem

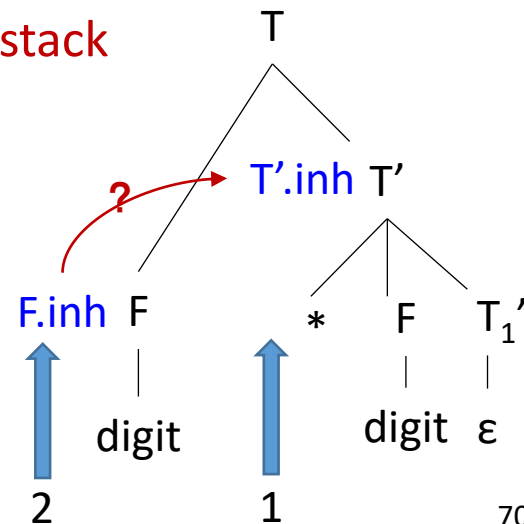


- It is not natural to evaluate inherited attributes
 - ◆ Example: how to get $T'.inh$
- Claims: inherited attributes are on the stack
 - ◆ L-SDD guarantee they've already been computed
 - ◆ But computed by previous productions - **deep in the stack**

• Solution

- ◆ **Hack the stack** to dig out those values

- (1) $T \rightarrow F$ $\{T'.inh = F.val\}$ $T' \{T.val = T'.syn\}$
 - (2) $T' \rightarrow *F$ $\{T_1'.inh = T'.int \times F.val\}$ $T_1' \{T'.syn = T_1'.syn\}$
 - (3) $T' \rightarrow \epsilon$ $\{T'.syn = T'.inh\}$
 - (4) $F \rightarrow \text{digit}$ $\{F.val = \text{digit.lexval}\}$



L-SDD in LR - Marker



- Given the following SDD, where $|\alpha| \neq |\beta|$
 - ◆ $A \rightarrow X\alpha \{Y.in = X.s\} Y \mid X\beta \{Y.in = X.s\} Y$
 - ◆ $Y \rightarrow \gamma \{Y.s = f(Y.in)\}$
- 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_1, M_2 right before Y
 - ◆ $A \rightarrow X\alpha M_1Y \mid X\beta M_2Y$
 - ◆ $M_1 \rightarrow \epsilon \{M_1.s = \text{stack}[\text{top} - |\alpha|].s\} \quad // M_1.s = X.s$
 - ◆ $M_2 \rightarrow \epsilon \{M_2.s = \text{stack}[\text{top} - |\beta|].s\} \quad // M_2.s = X.s$
 - ◆ $Y \rightarrow \gamma \{Y.s = f(\text{stack}[\text{top} - |\gamma|].s)\} \quad // Y.s = M_1.s \text{ or } Y.s = M_2.s$
- 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 $M \rightarrow \epsilon$ [空产生式]
 - Modify the action $\{a\}$ if marker M replaces it in some production $A \rightarrow \alpha \{a\} \beta$, and associate with $M \rightarrow \epsilon$ 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]
 - 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]

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

$A \rightarrow M BC$

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



Example



(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) $T' \rightarrow \epsilon \{T'.syn = T'.inh\}$
(4) $F \rightarrow digit \{F.val = digit.lexval\}$



(1) $T \rightarrow F \mathbf{M} T' \{T.val = T'.syn\}$
 $\mathbf{M} \rightarrow \epsilon \{\mathbf{M.i} = F.val; \mathbf{M.s} = M.i;\}$
(2) $T' \rightarrow *F \mathbf{N} T_1' \{T'.syn = T_1'.syn\}$
 $\mathbf{N} \rightarrow \epsilon \{\mathbf{N.i1} = T.inh; \mathbf{N.i2} = F.val; \mathbf{N.s} = N.i1 * N.i2\}$
(3) $T' \rightarrow \epsilon \{T'.syn = T'.inh\}$
(4) $F \rightarrow digit \{F.val = digit.lexval\}$

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



$A \rightarrow \mathbf{M} B C$

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



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) $T' \rightarrow \varepsilon \{T'.syn = T'.inh\}$
(4) $F \rightarrow digit \{F.val = digit.lexval\}$

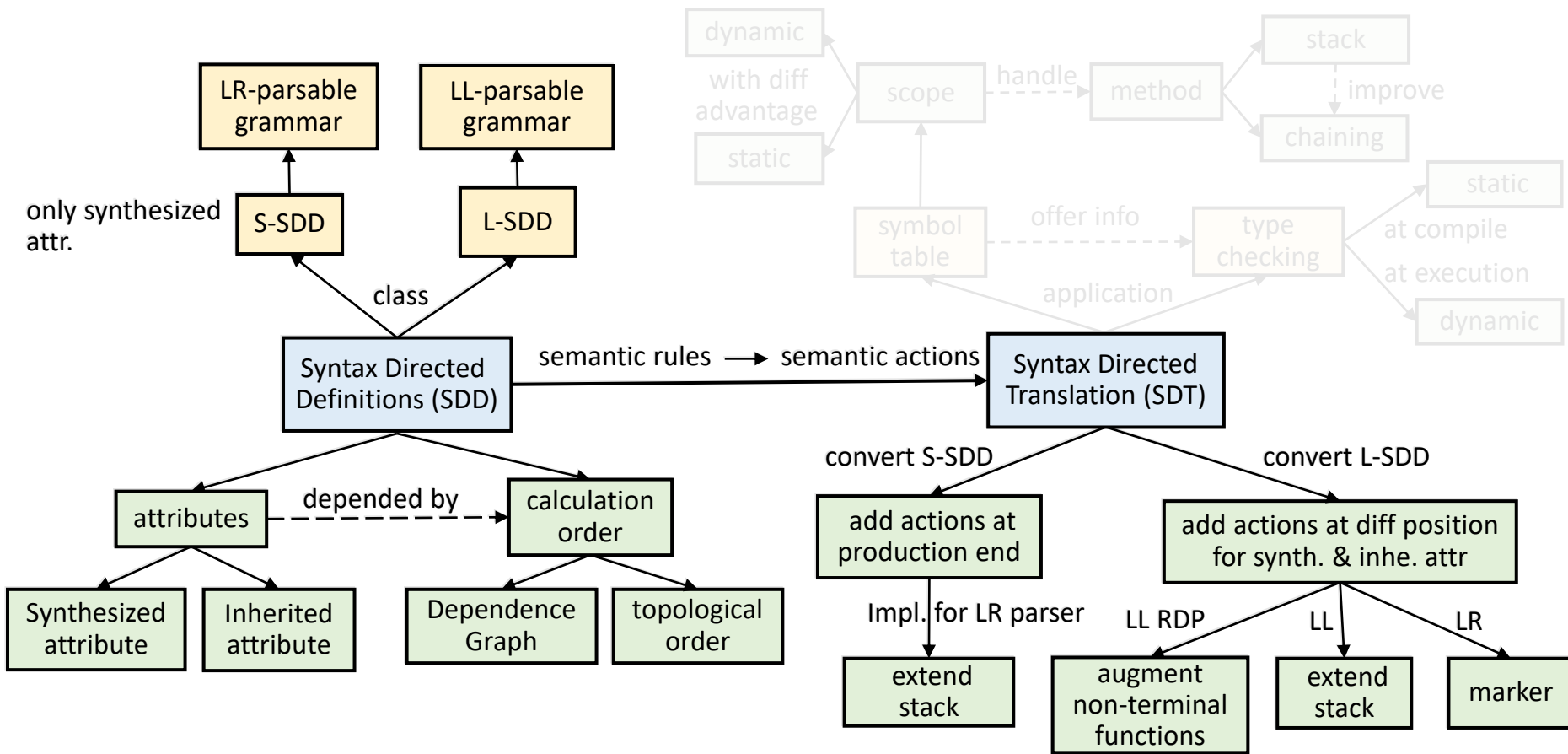
(1) $T \rightarrow F M T' \{T.val = T'.syn\}$
 $M \rightarrow \varepsilon \{M.i = F.val; M.s = M.i;\}$
(2) $T' \rightarrow *F N T_1' \{T'.syn = T_1'.syn\}$
 $N \rightarrow \varepsilon \{N.i1 = T.inh; N.i2 = F.val; N.s = N.i1 * N.i2\}$
(3) $T' \rightarrow \varepsilon \{T'.syn = T'.inh\}$
(4) $F \rightarrow digit \{F.val = digit.lexval\}$



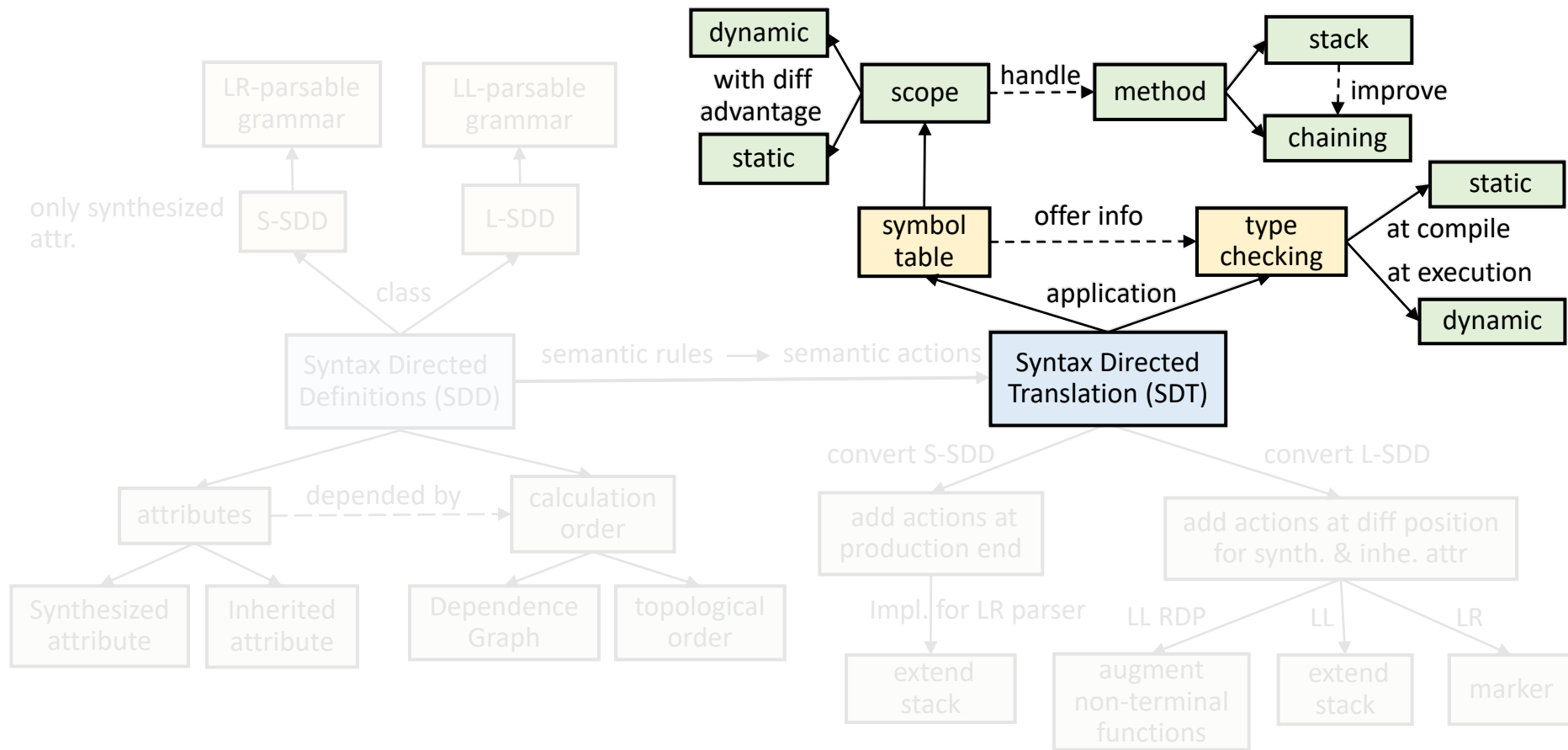
(1) $T \rightarrow F M T' \{stack[top-2].val = stack[top].syn; top = top - 2;\}$
 $M \rightarrow \varepsilon \{stack[top+1].T'.inh = stack[top].val; top = top + 1;\}$
(2) $T' \rightarrow *F N T_1' \{stack[top-3].syn = stack[top].syn; top = top - 3;\}$
 $N \rightarrow \varepsilon \{stack[top+1].T'.inh = stack[top-2].T'.inh * stack[top].val; top = top + 1;\}$
(3) $T' \rightarrow \varepsilon \{stack[top+1].syn = stack[top].T'.inh; top = top + 1;\}$
(4) $F \rightarrow digit \{stack[top].val = stack[top].lexval;\}$



Content



Content



Binding [绑定]

- **Binding**: matching identifier use with definition[使用-定义]
 - ◆ Definition: associating an id with a memory location
 - ◆ Binding is an essential step before machine code generation
- If there are multiple definitions, which one to use?

```
void foo()  
{  
    char x; /* allocated at mem[0x100] */  
    ...  
    {  
        int x; /* allocated at mem[0x200] */  
        ...  
    }  
    x = x + 1; /* add mem[0x100],1 ? add mem[0x200],1 ?  
}
```

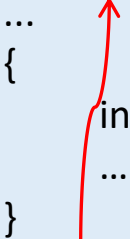
Scope [作用域]

- **Scope**: **program region** where a definition can be bound
 - ◆ Uses of identifier in the scope is bound to that definition
 - ◆ For Java: private, global, static, etc...
- Some properties of scopes
 - ◆ Use not in scope of any definition results in **undefined error**
 - ◆ Scopes for the same identifier can **never overlap**
 - There is **at most one binding** at any given time
- Two types: static scoping and dynamic scoping [静态/动态作用域]
 - ◆ Depending on how scopes are formed

Static Scoping [静态作用域]

- Scopes formed by where definitions are in **program text**[声明起作用的那段区域]
 - ◆ Also known as **lexical scoping** since related to program text C/C++, Java, Python, JavaScript[也叫词法作用域]
- Rule: bind to the **closest enclosing definition**

```
void foo()  
{  
    char x;  
    ...  
    {  
        int x;  
        ...  
    }  
    x = x + 1;  
}
```



Dynamic Scoping[动态作用域]

- Scopes formed by when definitions happen **during runtime**[运行时决定]
 - ◆ Perl, Bash, LISP, Scheme
- Rule: bind to **most recent definition** in current execution

```
void foo()
{
    (1) char x;
    (2) if (...) {
    (3)   int x;
    (4)   x=x+1;
        }
    (5) x='a';
}
```

- Which x's definition is the most recent?
 - ◆ Execution (a): ...**(1)**...(2)...(5)
 - ◆ Execution (b): ...(1)...(2)...**(3)**...(4)...(5)

Static vs. Dynamic Scoping[对比]

- Most languages that started with dynamic scoping (LISP, Scheme, Perl) added static scoping afterwards
- Why? With **dynamic scoping**...
 - ◆ All bindings are done at execution time
 - ◆ Hard to figure out by eyeballing, for both compiler and human
- Pros of **static scoping**[静态的好处]
 - ◆ Static scoping leads to **fewer programmer errors**
 - Bindings readily apparent from lexical structure of code
 - ◆ Static scoping leads to **more efficient** code
 - Can determine bindings at compile time – allow code optimization

What is Symbol Table[符号表]

- **Symbol**: same thing as **identifier** (used interchangeably)
- **Symbol table**: a data structure that tracks info about all symbols
 - ◆ Each entry represents a definition of that identifier[标识符的定义]
 - ◆ Maintains definitions that reach current program point[当前作用域中的定义]
 - ◆ List updated whenever scopes are entered or exited [随作用域改变而更新]
 - ◆ Used to perform binding of identifier uses [用于绑定变量定义与使用]
 - Traversing the parse tree in a separate pass after parsing [单独扫描语法树]
 - Using semantic actions as an integral part of parsing pass [或与语义分析同时]
 - ◆ Usually discarded after generating executable binary
 - Machine code instructions no longer contain symbols
 - ◆ For use in debuggers, symbol tables may be included
 - To display symbol names instead of addresses in debuggers
 - For GCC, using '`gcc -g`' includes debug symbol tables

Maintaining Symbol Table[维护]

- Basic idea:

```
int x=0; ... void foo() { int  x=0; ... x=x+1; } ... x=x+1 ...
```

- ◆ Start processing *foo*:

- ▣ Add definition of *x*, overriding old definition of *x* if any

- ◆ After processing *foo*:

- ▣ Remove definition of *x*, restoring old definition of *x* if any

- Operations:

- ◆ *enter_scope()* start a new scope
 - ◆ *exit_scope()* exit current scope
 - ◆ *find_symbol(x)* find the information about *x*
 - ◆ *add_symbol(x)* add a symbol *x* to the symbol table
 - ◆ *check_symbol(x)* true if *x* is defined in current scope

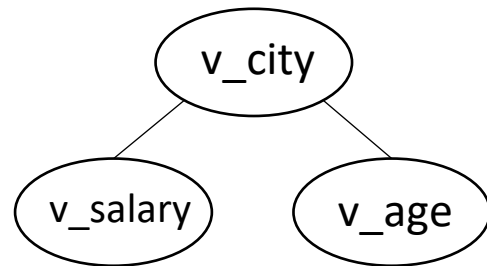
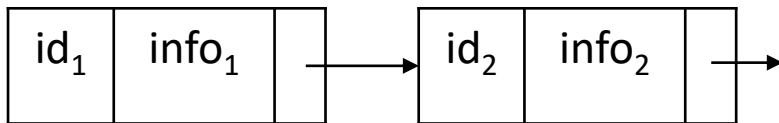
Symbol Table Structure [结构]

- **Front-end time** is affected by **symbol table access time**[符号表访问时间影响编译前端性能]
 - ◆ Front-end: lexical, syntax, semantic analysis
 - ◆ Frequent searches on any large data structure is expensive
 - ◆ Symbol table design is important for compiler performance
- What **data structure** to choose? [可选数据结构]
 - ◆ **List**[线性表]
 - ◆ **Binary tree**[二叉树]
 - ◆ **Hash table**[哈希表]
- Tradeoffs: **time** vs. **space**[空间和时间的权衡]
 - ◆ Let us first consider the organization without scope

Symbol Table Structure (cont.)

- **Array:** no space wasted, insert/delete: $O(n)$, search: $O(n)$
- **Linked list:** extra pointer space, insert/delete: $O(1)$, search: $O(n)$
 - ◆ Optimization: move recently used identifier to the head
 - ◆ Frequently used identifiers are found more quickly
- **Binary tree:** use more space than array/list
 - ◆ But insert/delete/search is $O(\log n)$ on balanced tree
 - ◆ In the worst case, tree may reduce to linked list
 - ▢ Then insert/delete/search becomes $O(n)$

| | |
|--------|----------|
| id_1 | $info_1$ |
| id_2 | $info_2$ |
| | |



SDD Implementation Revisit



- S-SDD & LR: Slack manipulation
- L-SDD: Remove Left Recursion
 - L-SDD & RDP: Extend RDP Function + Slack manipulation

◆ **Inherited**: arguments; **Synthesized**: return

◆ Use local variables[增加局部变量]

◆ Embed semantic actions[嵌入语义动作]

◆ Inherited: along with the symbol; Synthesized: right below the symbol

1. 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

- L-SDD & LL(1): Slack manipulation
- L-SDD & LR: Marker + Slack manipulation

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



A \rightarrow M BC

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

A \rightarrow X α {Y.in= X.s} Y | X β {Y.in= X.s} Y

Y \rightarrow γ {Y.s= f(Y.in)}

A \rightarrow X α M₁ Y | X β M₂ Y

M₁ \rightarrow ϵ {M₁.s= stack[top-| α |].s} // M₁.s= X.s

M₂ \rightarrow ϵ {M₂.s= stack[top-| β |].s} // M₂.s= X.s

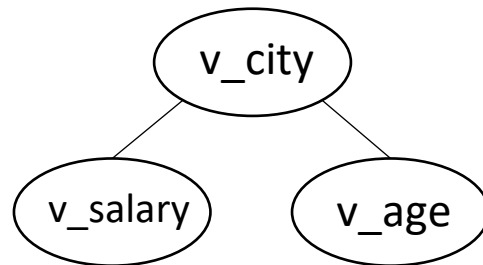
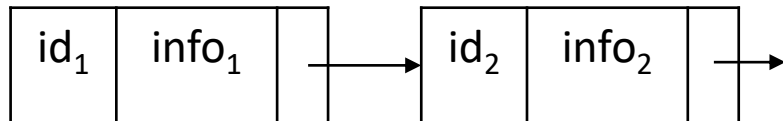
Y \rightarrow γ {Y.s= f(stack[top-| γ |].s)} // Y.s= M₁.s or Y.s= M₂.s



Symbol Table Revisit

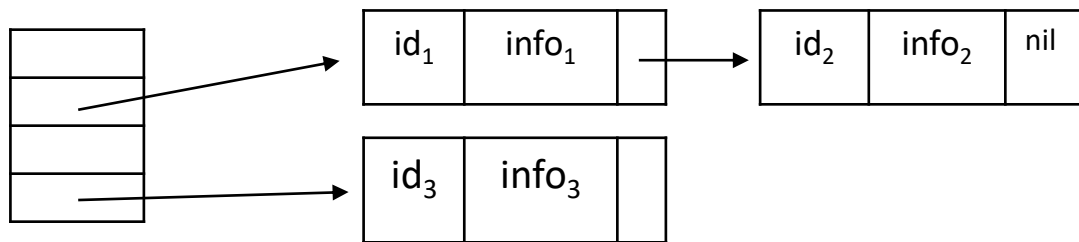
- **Binding:** matching **identifier use** with **definition**[使用-定义]
- **Scoping:**
 - **Static:** Can be obtained by static code analysis
 - **Dynamic:** becomes clear only during execution
- **Symbol:** same thing as **identifier**
- **Symbol table:** a **data structure** that tracks info about all symbols
- **Constructing Symbol table:** a tradeoff between **time** and space
 - Array, Linked list, Binary tree, **Hash table**

| | |
|-----------------|-------------------|
| id ₁ | info ₁ |
| id ₂ | info ₂ |
| | |



Symbol Table Structure (cont.)

- **hash(id_name) → index**[哈希表]
 - ◆ A **hash function** decides **mapping** from identifier to index
 - ◆ Conflicts resolved by chaining multiple IDs to same index
- Memory consumption from hash table (**$N \ll M$**)
 - ◆ M: the size of hash table
 - ◆ N: the number of stored identifiers
- But insert/delete/search in **$O(1)$** time
 - ◆ Can become **$O(n)$** with frequent conflicts and long chains
- Most compilers choose hash table for its quick access time



Adding Scope to Symbol Table[作用域]

- To handle **multiple scopes** in a program [处理多个作用域]

- ◆ Conceptually, need **an individual table** for **each scope**

- In order to be able to **enter and exit scopes**

- ◆ Sometimes symbols in scope can be discarded on exit:

```
if (...) { int v; } /* block scope */  
/* v is no longer valid */
```

- ◆ Sometimes not:

```
class X { ... void foo() {...} ... } /* class scope */  
/* foo() is no longer valid */  
X v;  
call v.foo(); /* v.foo() is still valid */
```

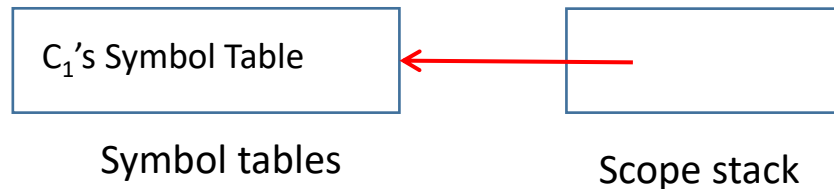
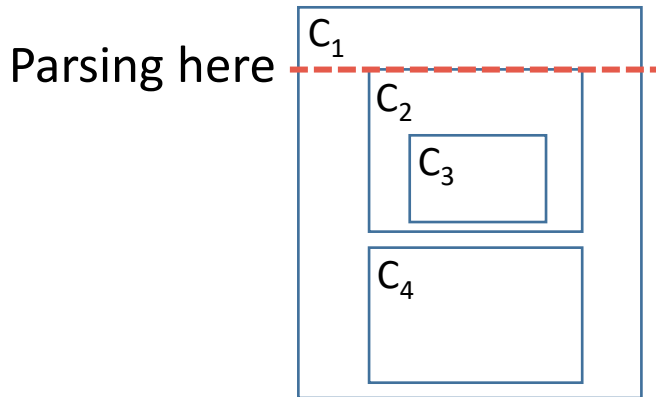
- How can scoping be enforced without discarding symbols?

- ◆ Keep **a stack of active scopes** at a given point

- ◆ Keep **a list of all reachable scopes** in the entire program

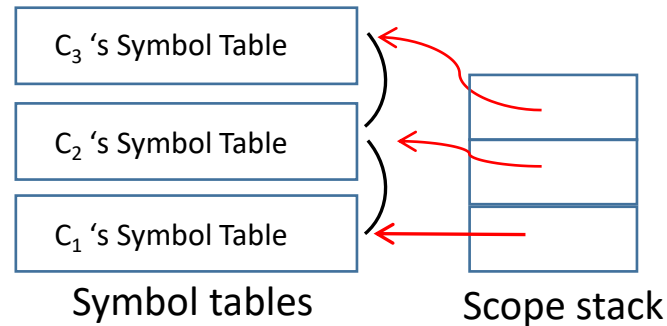
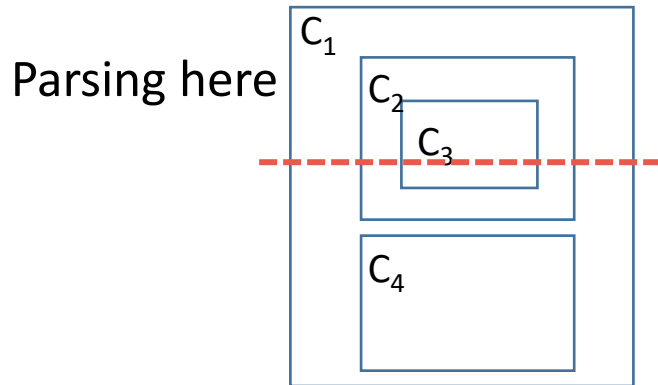
Handle Scopes with Stack

- Organize all symbol tables into **a scope stack**[作用域栈]
 - ◆ An individual symbol table for each scope
 - Scope is defined by **nested lexical structure**, e.g., $\{ C_1 \{ C_2 \{ C_3 \} \} \{ C_4 \} \}$
 - ◆ Stack holds one entry for each open scope
 - **Inner-most** scope is stored at the **top of the stack**
- Stack **push/pop** happen when **entering/exiting** a scope



Handle Scopes with Stack (cont.)

- Operations:
 - ◆ When **entering** a scope
 - Create a **new symbol table** to hold all variables declared in that scope
 - **Push** a **pointer to the symbol table** on the stack
 - ◆ **Pop** the pointer to the symbol table **when exiting scope**
 - ◆ Search from the top of the stack



Info Stored in Symbol Table

- Entry in symbol table
 - ◆ **String**: the name of identifier
 - ◆ **Kind**: variable, function, struct type, class type

| | | |
|--------|------|------------|
| string | kind | attributes |
|--------|------|------------|

- Attributes vary with the kind of symbols
 - ◆ **variable**: type, address of variable
 - ◆ **function**: prototype, address of function body
 - ◆ **struct type**: field names, field types
 - ◆ **class type**: symbol table for class

Attribute List in Symbol Table

- Type info can be arbitrarily complicated

- ◆ Type can be an array with multiple dimensions

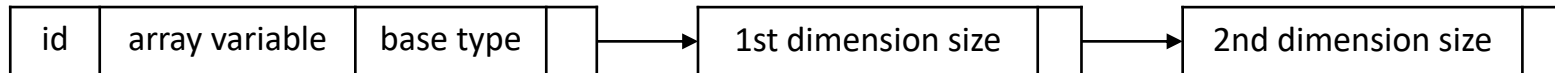
```
char arr[20][20];
```

- ◆ Type can be a struct with multiple fields

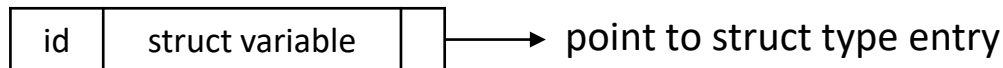
```
struct Point {  
    float x;  
    float y;  
} point;
```

- Store all type info in an **attribute list**

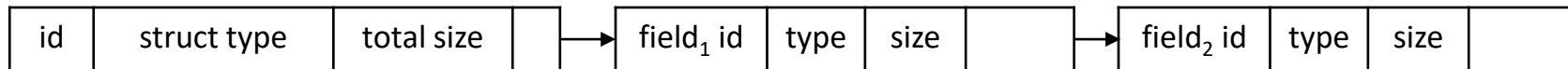
- ◆ Entry for an array variable with 2 dimensions



- ◆ Entry for a struct variable



- ◆ Entry for a struct type with 2 fields



Use Type Information[类型信息]

- Each variable or function entry contains **type** info
- Type info is used in later **code generation** stage[代码生成]
 - ◆ To calculate how much memory to alloc for a variable
 - Should a variable assignment be a **4 byte or 8 byte** copy?
 - ◆ To translate uses of variables to machine instructions
 - Should a '+' on variable be an **integer or a floating** point add?
 - ◆ To translate calls to functions to machine instructions
 - What are the **types of arguments and return value** of the function?
- Also used in later **code optimization** stage[代码优化]
 - ◆ To help compiler understand semantics of program
- Also used in **semantic analysis** stage for **Type Checking**
 - ◆ Uses types to check semantic correctness of program

Type and Type Checking

- **Type**: a set of **values** and a set of **operations** on these values
- **Type checking**: verifying **type consistency** across program
 - ◆ A program is type consistent if all operators are consistent with the operand value types
 - ◆ Much of what we do in semantic analysis is type checking
- Some type checking examples:
 - ◆ Given **char *str = "Hello";**
 - ▢ **str[2]** is consistent: **char*** type allows **[]** operator
 - ▢ **str/2** is not: **char*** type does not allow **/** operator
 - ◆ Given **int pi = 3;**
 - ▢ **pi/2** is consistent: **int** type allows **/** operator
 - ▢ **pi=3.14** is not: **=** operator not allowed on different types
 - ▢ Compiler must **type convert implicitly** to make it consistent

Static Type Checking[静态类型检查]

- **Static type checking** at compile time
 - ◆ **Infers**[推断] whether the program is type consistent through code analysis
 - Collect info via declarations that store in symbol table
 - Check the types involved in each operation
 - E.g., `int a, b, c; a = b + c;` can be proven type consistent because the addition of two *ints* is still an *int*
- Difficult for a language to only do static type checking
 - ◆ Some type errors usually cannot be detected at compile time
 - E.g., `a` and `b` are of type *int*, `a * b` may not in the **valid range** of *int*
 - **Typecasting** can be a pretty risky thing to do - Basically, typecast suspends type checking
`unsigned a; (int) a;`

Dynamic Type Checking[动态检查]

- **Dynamic type checking** at execution time
 - ◆ Type consistency by checking types of **runtime values**
 - ◆ Include type info for each data location at runtime
 - E.g., a variable of type double would contain both the **actual double value** and some kind of **tag indicating “double type”**
 - The execution of any operation begins by first checking these type tags
 - The operation is performed only if everything checks out
 - Otherwise, a type error occurs and usually halts execution
 - ◆ Array bounds check:
 - Is **int A[10], i; ... A[i] = i;** type consistent?
- Static type checking is always more desirable. Why?
 - ◆ Always good to catch more errors before runtime
 - ◆ Dynamic type checking carries runtime overhead

Static vs. Dynamic Typing[静态-动态]

- Static typing: C/C++, Java, ...
 - ◆ Variables have **static types** → hold only one type of value
 - E.g. `int x;` → x can only hold ints
 - E.g. `char *x;` → x can only hold char pointers
 - ◆ How are types assigned to variables?
 - C/C++, Java: types are explicitly defined
 - `int x;` → explicit assignment of type int to x
- Pros/cons of static typing
 - ◆ **More programmer effort**
 - Programmer must adhere to strict type rules
 - Defining advanced types can be quite complex (e.g. classes)
 - ◆ **Less bugs and execution time**
 - Thanks to static type checking

Static vs. Dynamic Typing (cont.)

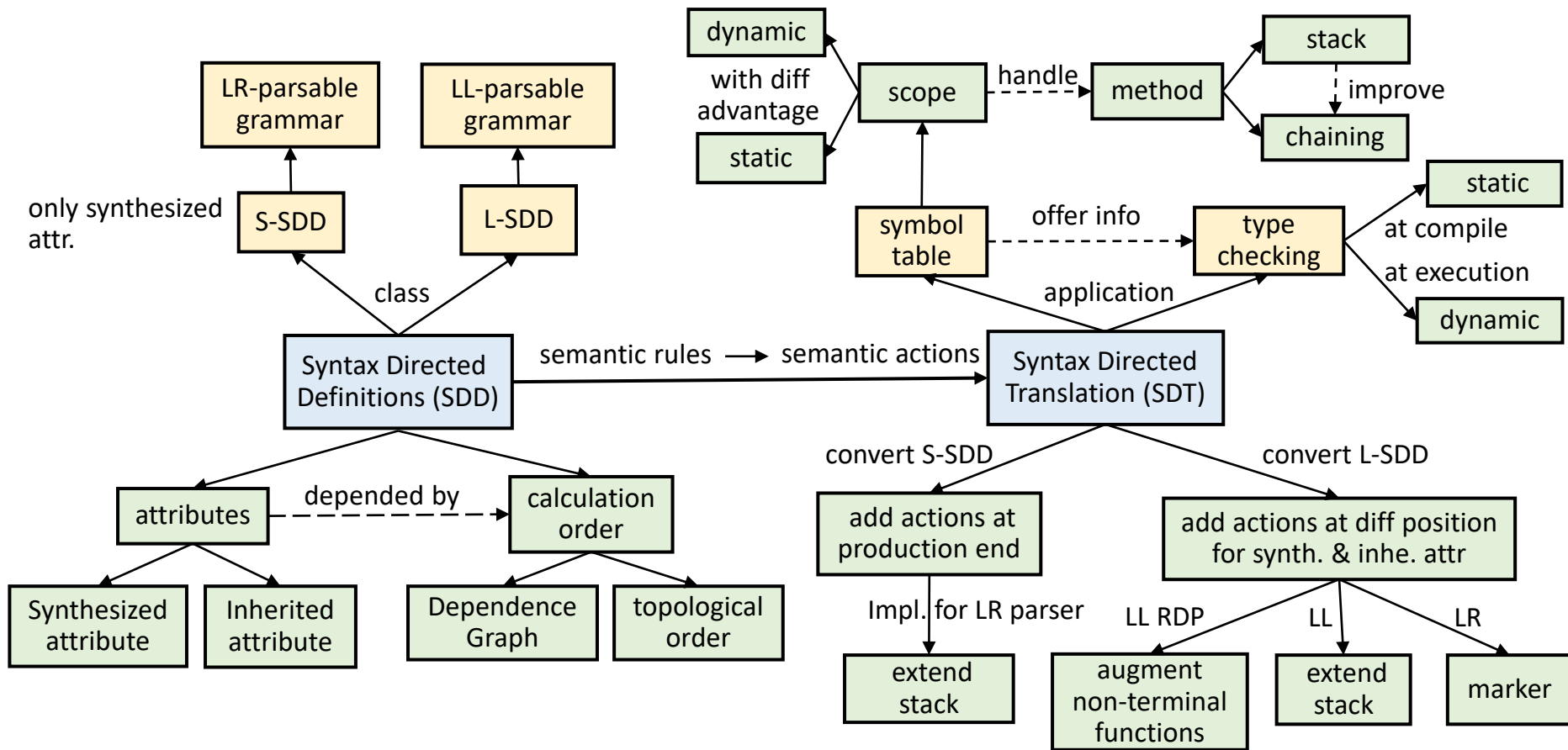
- Dynamic Typing: Python, JavaScript, PHP, ...
 - ◆ Variables have dynamic types → can hold multiple types

```
var x; /* var declaration without a static type */  
x = 1; /* now x holds an integer value */  
x = "one"; /* now x holds a string value */
```
 - ◆ How are types assigned to variables?
 - Type is a runtime property → type tags stored with values
 - Dynamic type checking must be done during runtime
- Pros/cons of dynamic typing
 - ◆ Less programmer effort
 - Flexible type rule means program is **more malleable**[可塑性强]
 - Absence of types declarations means **shorter code**
 - Suitable for scripting or prototyping languages
 - ◆ More program bugs and execution time
 - Due to **dynamic type checking**

Type System[类型系统]

- **Static/dynamic typing** are **type systems**
 - ◆ Type System: **types** and **type rules** of a language
- **Static/dynamic type checking** are **methods**
 - ◆ Methods to **enforce the rules** of the given type system
- Static type checking is not used only for static typing
 - ◆ Also used for dynamic typing **if types can be inferred** at compile time
- Dynamic type checking is not used only for dynamic typing
 - ◆ Some features of statically typed languages require it
 - e.g., **downcasting**[父类向子类的强制类型转换]

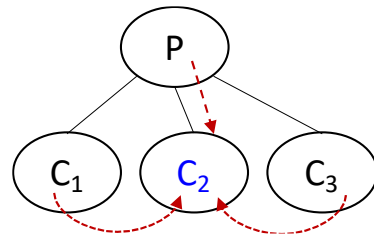
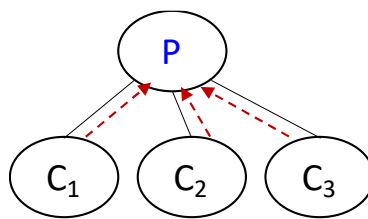
Summary



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



Summary

- Static and Dynamic Scoping
 - ◆ Pros of **static**: fewer errors + more efficient
- Symbol and Symbol Table
 - ◆ Structuring: array, list, binary tree, **hash table**
 - ◆ Trade-off between **time** and **space**
- Scope handling
 - ◆ Stack: Inefficient searching and use of memory
 - ◆ So **chaining** is introduced
- Static and dynamic typing and type checking
 - ◆ Static: **less bugs and execution time**, but **more effort**
 - ◆ Dynamic: **less effort**, but **more bugs and execution time**
 - ◆ Static ones are more desirable, but is often insufficient, e.g., downcasting

Further Reading



- Dragon Book, 2nd 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.





编译原理

Compiler Principles

Exercise

Syntax and Semantic Analysis

赵帅

计算机学院
中山大学

Exercise



- 对于一个无二义性文法G，它的一个句子的最左与最右推导产生的分析树一定是一样的吗？
 - 对的，是这样的。语法树会隐去推导顺序
- 怎么样判断一个文法是否为LL(1)型文法？
 - 对文法中的每个产生式 $A \rightarrow \alpha \mid \beta$:
 - $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$.
 - If $\beta \Rightarrow \epsilon$, then $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$.
- LL(1)语法分析表行数与列数分别由哪些因素确定？
 - 行数是该文法的非终结符个数，列数为终结符个数+1（结束符号\$）
- 设句型aaAb的最右推导为 $S \Rightarrow aBb \Rightarrow aaAb$ ，句柄为aA，该句型的活前缀是？
 - a, aa, aaA



Exercise

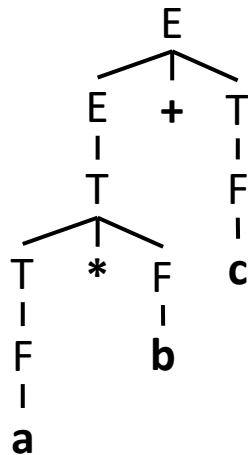


- 找出以下文法中的句型的所有短语，直接短语，句柄

- 文法: $G(E): E \rightarrow T \mid E+T; \quad T \rightarrow F \mid T*F; \quad F \rightarrow (E) \mid a \mid b \mid c$

- 句型: $a * b + c$

- 第一步：画出语法树
- 第二步：找出语法树的叶子节点
 - $a, *, b, +, c$
- 第三步：由叶子节点找出短语
 - $a, a*b, b, c, a*b+c$ (why not $*b, b+c, *, +?$)
- 第四步：从短语中找出直接短语
 - a, b, c
- 第五步：从直接短语中找出句柄
 - a



Exercise



- 给定一下文法G(S), 请构建该文法的**LL(1)**分析表

- $S \rightarrow A$
- $A \rightarrow \varepsilon$
- $A \rightarrow bbA$

| | b | \$ |
|---|---|----|
| S | | |
| A | | |

- FIRST & Follow:**

- $S \rightarrow A$: $\text{FIRST}(\alpha) = \{b, \varepsilon\}$
- $A \rightarrow \varepsilon$: $\text{FIRST}(\alpha) = \{\varepsilon\}$
- $A \rightarrow bbA$: $\text{FIRST}(\alpha) = \{b\}$
- $\text{FOLLOW}(S) = \{\$ \}$
- $\text{FOLLOW}(A) = \{\$, \varepsilon\}$

| | b | \$ |
|---|---------------------|----|
| S | $S \rightarrow A$ | |
| A | $A \rightarrow bbA$ | |

use FIRST

- Parse Table:**

| | b | \$ |
|---|---------------------|-----------------------------|
| S | $S \rightarrow A$ | $S \rightarrow A$ |
| A | $A \rightarrow bbA$ | $A \rightarrow \varepsilon$ |

use FOLLOW

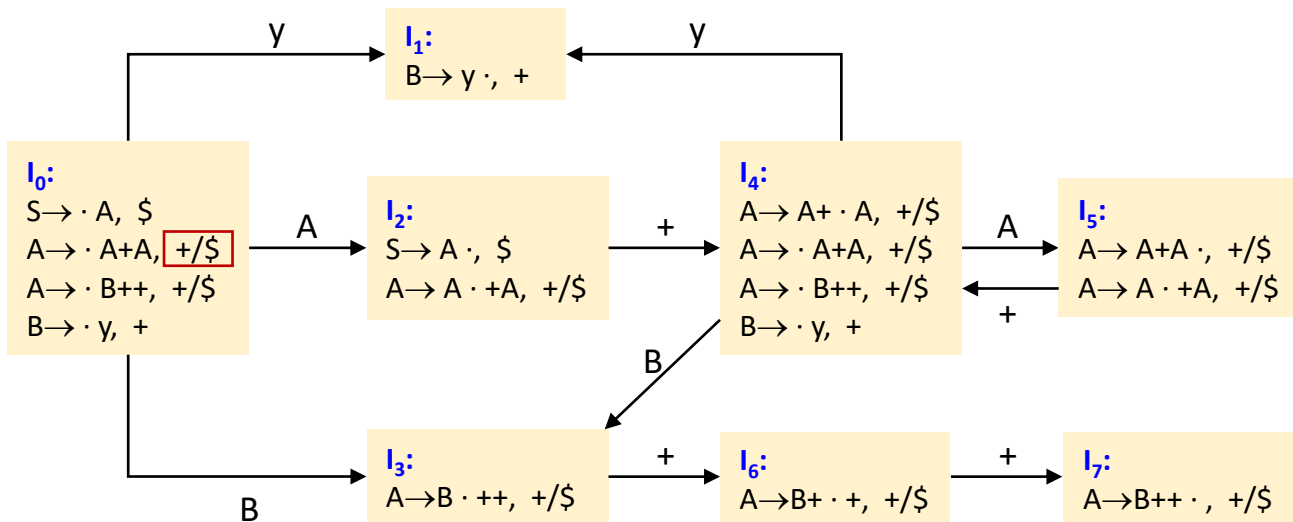


Exercise



• 给定一下文法G(S), 请构建识别活前缀的DFA与LR(1)分析表

- $S \rightarrow A$
- $A \rightarrow A + A \mid B ++$
- $B \rightarrow y$

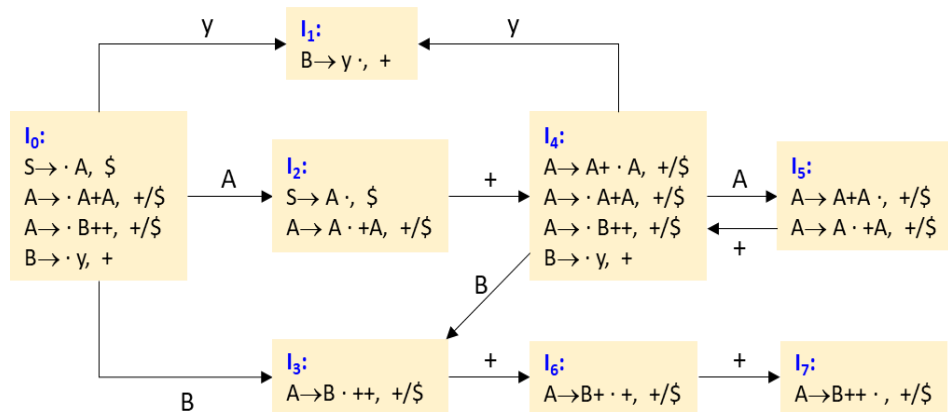


Exercise



• 给定一下文法G(S)，请构建识别活前缀的DFA与LR(1)分析表

- (1) $S \rightarrow A$
- (2) $A \rightarrow A + A$
- (3) $A \rightarrow B ++$
- (4) $B \rightarrow y$



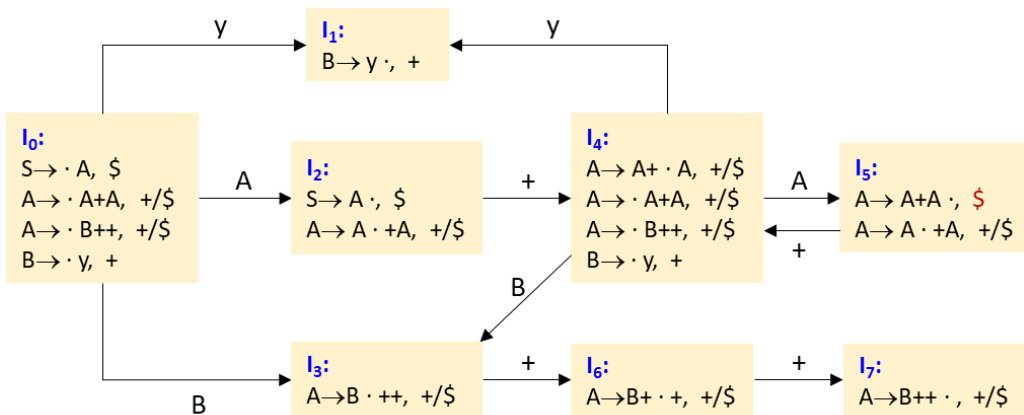
| S | ACTION | | | GOTO | |
|---|--------|-------|-----|------|---|
| | y | + | \$ | A | B |
| 0 | s1 | | | 2 | 3 |
| 1 | | r4 | | | |
| 2 | | s4 | acc | | |
| 3 | | s6 | | | |
| 4 | s1 | | | 5 | 3 |
| 5 | | r2/s4 | r2 | | |
| 6 | | s7 | | | |
| 7 | | r3 | r3 | | |



Exercise



- 该文法是LR(1)型文法吗? 若不是, 应如何解决? (有多种选择)



| S | ACTION | | | GOTO | |
|---|--------|----|-----|------|---|
| | y | + | \$ | A | B |
| 0 | s1 | | | 2 | 3 |
| 1 | | r4 | | | |
| 2 | | s4 | acc | | |
| 3 | | s6 | | | |
| 4 | s1 | | | 5 | 3 |
| 5 | | s4 | r2 | | |
| 6 | | s7 | | | |
| 7 | | r3 | r3 | | |



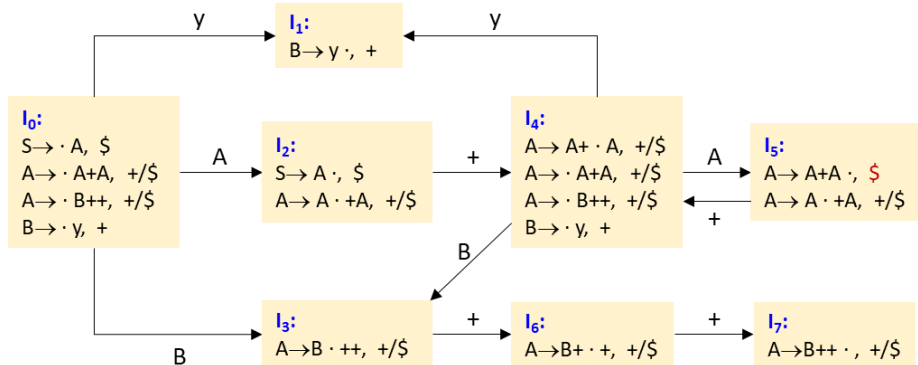
Exercise



- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

| Stack | Input | ACTION | GOTO |
|----------|-------|---------------|------|
| <u>0</u> | \$ | <u>y</u> ++\$ | |



| S | ACTION | | | GOTO | |
|---|--------|----|-----|------|---|
| | y | + | \$ | A | B |
| 0 | s1 | | | 2 | 3 |
| 1 | | r4 | | | |
| 2 | | s4 | acc | | |
| 3 | | s6 | | | |
| 4 | s1 | | | 5 | 3 |
| 5 | | r4 | r2 | | |
| 6 | | s7 | | | |
| 7 | | r3 | r3 | | |

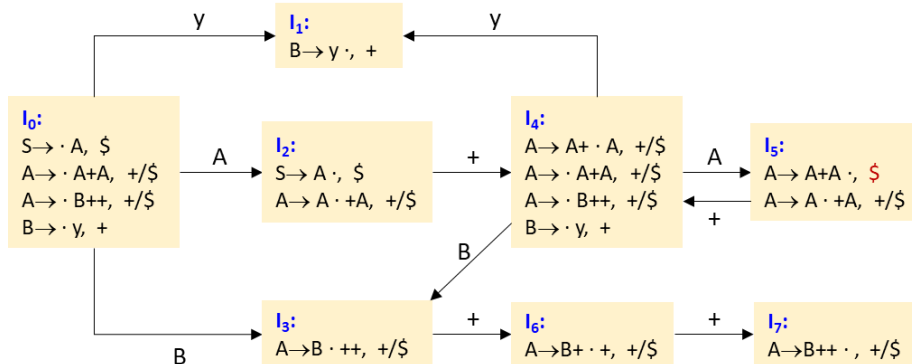
Exercise



- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

| Stack | Input | ACTION | GOTO |
|----------|-------|---------------|------|
| <u>0</u> | \$ | <u>y</u> ++\$ | s1 |



| S | ACTION | | | GOTO | |
|---|--------|----|-----|------|---|
| | y | + | \$ | A | B |
| 0 | s1 | | | 2 | 3 |
| 1 | | r4 | | | |
| 2 | | s4 | acc | | |
| 3 | | s6 | | | |
| 4 | s1 | | | 5 | 3 |
| 5 | | r4 | r2 | | |
| 6 | | s7 | | | |
| 7 | | r3 | r3 | | |

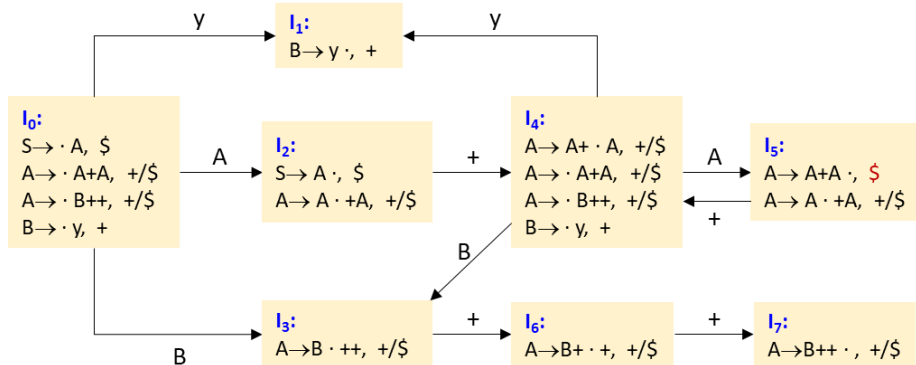
Exercise



- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

| Stack | Input | ACTION | GOTO |
|------------|-------|---------------|------|
| <u>0</u> | \$ | <u>y</u> ++\$ | s1 |
| 0 <u>1</u> | \$y | <u>+</u> +\$ | |



| S | ACTION | | | GOTO | |
|---|--------|-------------------------------------|-----|------|---|
| | y | + | \$ | A | B |
| 0 | s1 | | | 2 | 3 |
| 1 | | r4 | | | |
| 2 | | s4 | acc | | |
| 3 | | s6 | | | |
| 4 | s1 | | | 5 | 3 |
| 5 | | s4 | r2 | | |
| 6 | | s7 | | | |
| 7 | | r3 | r3 | | |

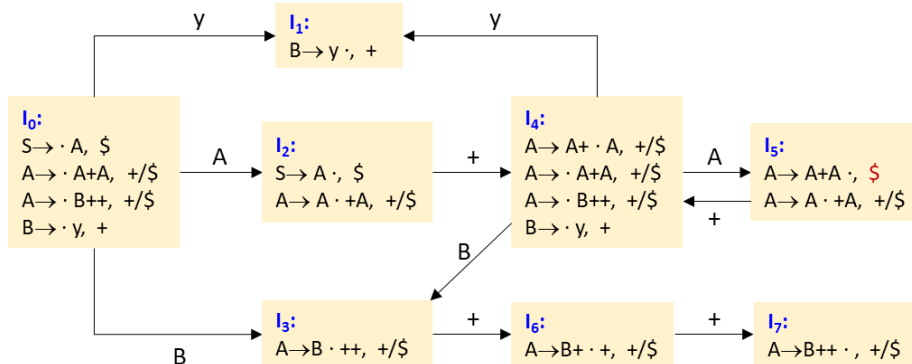
Exercise



- 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

| Stack | Input | ACTION | GOTO |
|------------|-------|---------------|-------------------------|
| <u>0</u> | \$ | <u>y</u> ++\$ | s1 |
| 0 <u>1</u> | \$y | <u>+</u> +\$ | r4($B \rightarrow y$) |



| S | ACTION | | | GOTO | |
|---|--------|----|-----|------|---|
| | y | + | \$ | A | B |
| 0 | s1 | | | 2 | 3 |
| 1 | | r4 | | | |
| 2 | | s4 | acc | | |
| 3 | | s6 | | | |
| 4 | s1 | | | 5 | 3 |
| 5 | | s4 | r2 | | |
| 6 | | s7 | | | |
| 7 | | r3 | r3 | | |

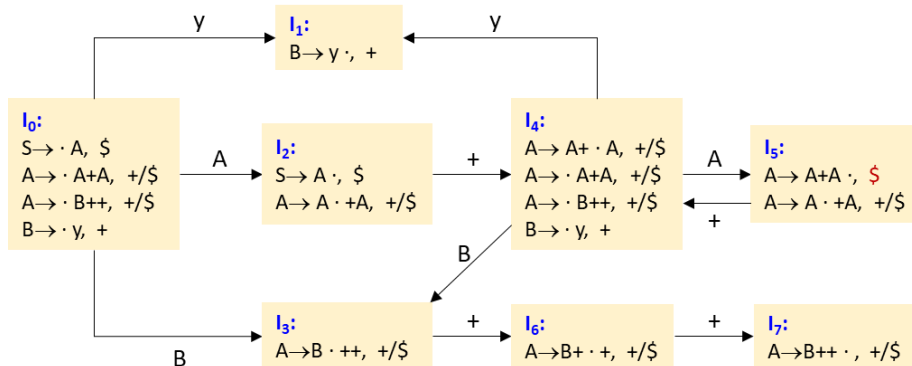
Exercise



• 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

| Stack | Input | ACTION | GOTO |
|------------|-------|---------------|-------------------------|
| <u>0</u> | \$ | <u>y</u> ++\$ | s1 |
| 0 <u>1</u> | \$y | <u>++</u> \$ | r4($B \rightarrow y$) |
| 0 | \$B | <u>++</u> \$ | 3 |



| S | ACTION | | | GOTO | |
|---|--------|----|-----|------|---|
| | y | + | \$ | A | B |
| 0 | s1 | | | 2 | 3 |
| 1 | | r4 | | | |
| 2 | | s4 | acc | | |
| 3 | | s6 | | | |
| 4 | s1 | | | 5 | 3 |
| 5 | | s4 | r2 | | |
| 6 | | s7 | | | |
| 7 | | r3 | r3 | | |

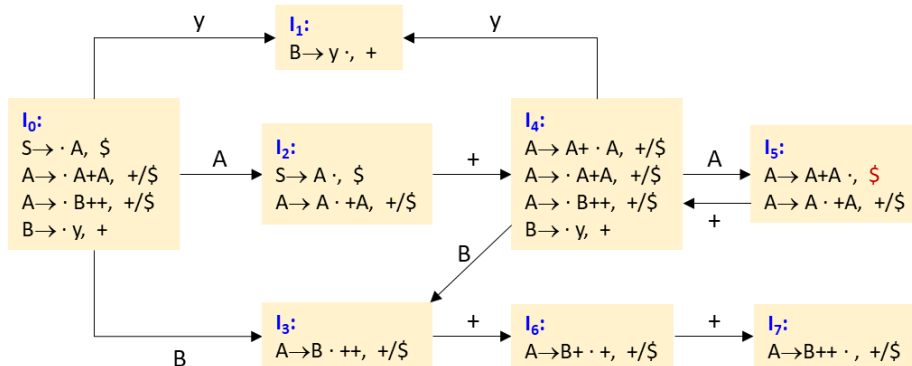
Exercise



• 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

| | Stack | Input | ACTION | GOTO |
|------------|-------|---------------|-------------------------|------|
| <u>0</u> | \$ | <u>y</u> ++\$ | s1 | |
| 0 <u>1</u> | \$y | <u>++</u> \$ | r4($B \rightarrow y$) | |
| 0 | \$B | <u>++</u> \$ | | 3 |
| 0 <u>3</u> | \$B | <u>++</u> \$ | s6 | |



| S | ACTION | | | GOTO | |
|---|--------|-------------------------------------|-----|------|---|
| | y | + | \$ | A | B |
| 0 | s1 | | | 2 | 3 |
| 1 | | r4 | | | |
| 2 | | s4 | acc | | |
| 3 | | s6 | | | |
| 4 | s1 | | | 5 | 3 |
| 5 | | s4 | r2 | | |
| 6 | | s7 | | | |
| 7 | | r3 | r3 | | |

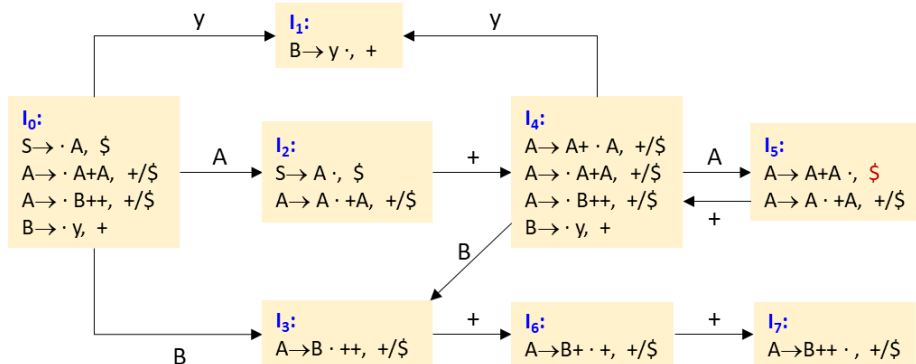
Exercise



• 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

| | Stack | Input | ACTION | GOTO |
|-------------|-------|---------------|-------------------------|------|
| <u>0</u> | \$ | <u>y</u> ++\$ | s1 | |
| 0 <u>1</u> | \$y | <u>++</u> \$ | r4($B \rightarrow y$) | |
| 0 | \$B | <u>++</u> \$ | | 3 |
| 0 <u>3</u> | \$B | <u>++</u> \$ | s6 | |
| 03 <u>6</u> | \$B+ | <u>+</u> \$ | s7 | |



| S | ACTION | | | GOTO | |
|---|--------|-------------------------------------|-----|------|---|
| | y | + | \$ | A | B |
| 0 | s1 | | | 2 | 3 |
| 1 | | r4 | | | |
| 2 | | s4 | acc | | |
| 3 | | s6 | | | |
| 4 | s1 | | | 5 | 3 |
| 5 | | s4 | r2 | | |
| 6 | | s7 | | | |
| 7 | | r3 | r3 | | |

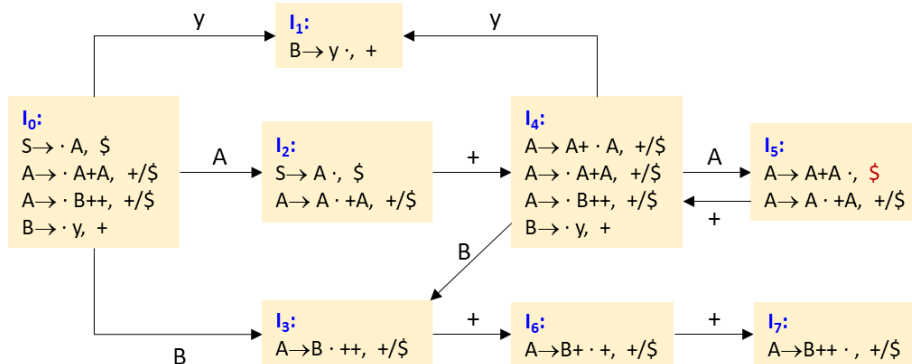
Exercise



• 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

| Stack | Input | ACTION | GOTO |
|--------------|-------|---------------|----------------------------|
| <u>0</u> | \$ | <u>y</u> ++\$ | s1 |
| 0 <u>1</u> | \$y | <u>++</u> \$ | r4($B \rightarrow y$) |
| 0 | \$B | <u>++</u> \$ | 3 |
| 0 <u>3</u> | \$B | <u>++</u> \$ | s6 |
| 03 <u>6</u> | \$B+ | <u>+</u> \$ | s7 |
| 036 <u>7</u> | \$B++ | <u>\$</u> | r3($A \rightarrow B ++$) |



| S | ACTION | | | GOTO | |
|---|--------|----|-----|------|---|
| | y | + | \$ | A | B |
| 0 | s1 | | | 2 | 3 |
| 1 | | r4 | | | |
| 2 | | s4 | acc | | |
| 3 | | s6 | | | |
| 4 | s1 | | | 5 | 3 |
| 5 | | r4 | r2 | | |
| 6 | | s7 | | | |
| 7 | | r3 | r3 | | |

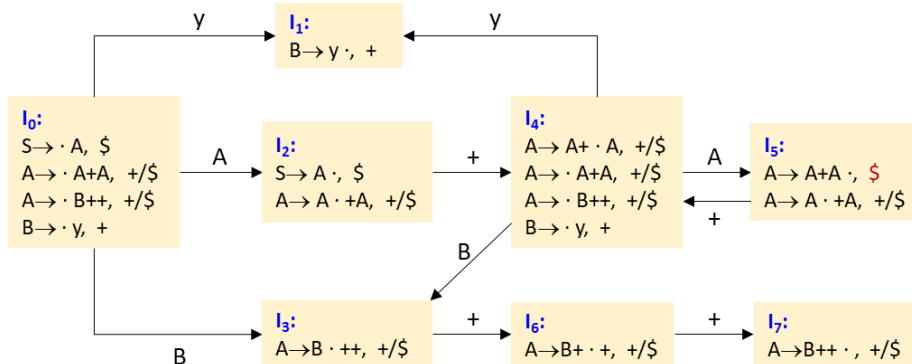
Exercise



• 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

| Stack | Input | ACTION | GOTO |
|--------------|-------|---------------|----------------------------|
| <u>0</u> | \$ | <u>y</u> ++\$ | s1 |
| 0 <u>1</u> | \$y | <u>++</u> \$ | r4($B \rightarrow y$) |
| 0 | \$B | <u>++</u> \$ | 3 |
| 0 <u>3</u> | \$B | <u>++</u> \$ | s6 |
| 03 <u>6</u> | \$B+ | <u>+</u> \$ | s7 |
| 036 <u>7</u> | \$B++ | <u>\$</u> | r3($A \rightarrow B ++$) |
| 0 | \$A | <u>\$</u> | 2 |



| S | ACTION | | | GOTO | |
|---|--------|----|-----|------|---|
| | y | + | \$ | A | B |
| 0 | s1 | | | 2 | 3 |
| 1 | | r4 | | | |
| 2 | | s4 | acc | | |
| 3 | | s6 | | | |
| 4 | s1 | | | 5 | 3 |
| 5 | | r4 | r2 | | |
| 6 | | s7 | | | |
| 7 | | r3 | r3 | | |

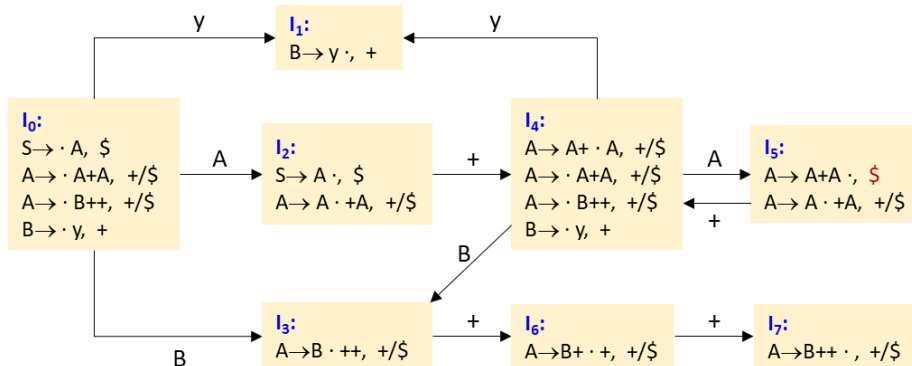
Exercise



• 对于句子 $y++$, 请给出其分析流程, 并说明 $y++$ 是否为该文法的一个句子?

(1) $S \rightarrow A$; (2) $A \rightarrow A + A$; (3) $A \rightarrow B ++$; (4) $B \rightarrow y$;

| Stack | Input | ACTION | GOTO |
|--------------|-------|---------------|----------------------------|
| <u>0</u> | \$ | <u>y</u> ++\$ | s1 |
| 0 <u>1</u> | \$y | <u>++</u> \$ | r4($B \rightarrow y$) |
| 0 | \$B | <u>++</u> \$ | 3 |
| 0 <u>3</u> | \$B | <u>++</u> \$ | s6 |
| 03 <u>6</u> | \$B+ | <u>+</u> \$ | s7 |
| 036 <u>7</u> | \$B++ | <u>\$</u> | r3($A \rightarrow B ++$) |
| 0 | \$A | <u>\$</u> | 2 |
| 0 <u>2</u> | \$A | <u>\$</u> | acc |



| S | ACTION | | | GOTO | |
|---|--------|----|-----|------|---|
| | y | + | \$ | A | B |
| 0 | s1 | | | 2 | 3 |
| 1 | | r4 | | | |
| 2 | | s4 | acc | | |
| 3 | | s6 | | | |
| 4 | s1 | | | 5 | 3 |
| 5 | | r4 | r2 | | |
| 6 | | s7 | | | |
| 7 | | r3 | r3 | | |

Exercise



给定文法G(S):

$$S \rightarrow Sb \mid bAa$$

$$A \rightarrow aSc \mid aSb \mid a$$

- (1) 拓广文法后，用LR(0)项目构造一个识别G(S)所有活前缀的DFA
- (2) 判断该文法是否为LR(0)文法，并阐明原因
- (3) 判断该文法是否为LR(1)文法，并阐明原因。
- (4) 试构造该文法的SLR(1)分析表。

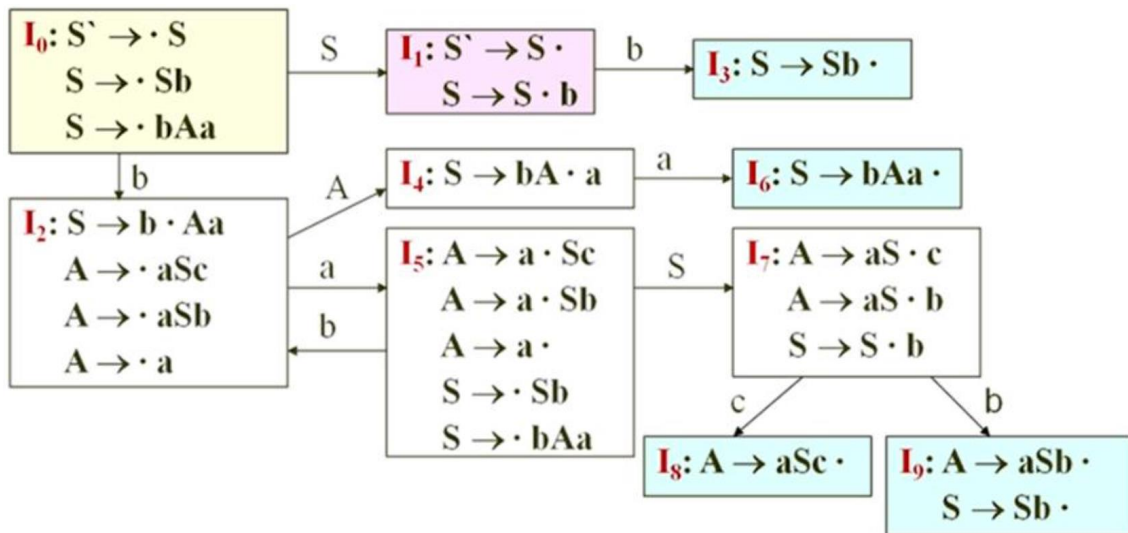
Exercise



给定文法G(S):

$$S \rightarrow Sb \mid bAa$$
$$A \rightarrow aSc \mid aSb \mid a$$

(1) 拓广文法后, 用LR(0)项目构造一个识别G(S)所有活前缀的DFA



Exercise



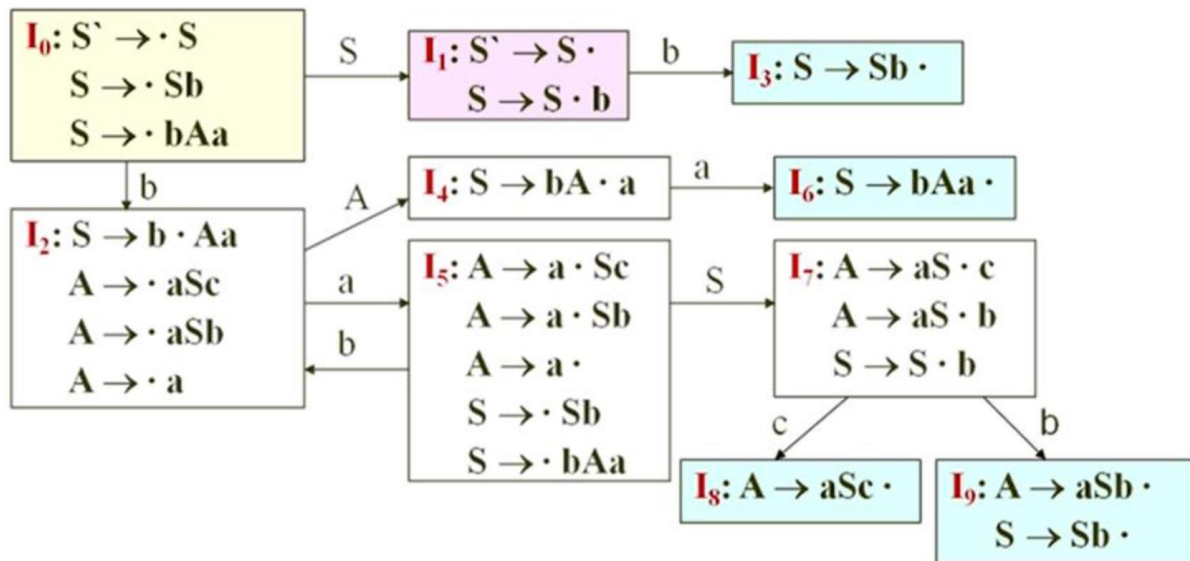
给定文法G(S):

$$S \rightarrow Sb \mid bAa$$

$$A \rightarrow aSc \mid aSb \mid a$$

(2) 判断该文法是否为LR(0)文法，并阐明原因

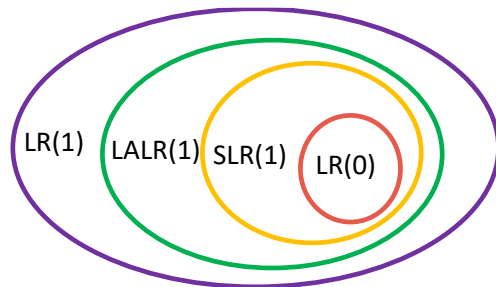
- 移进—归约: I_1, I_5
- 归约—归约: I_9



Exercise



给定文法G(S):

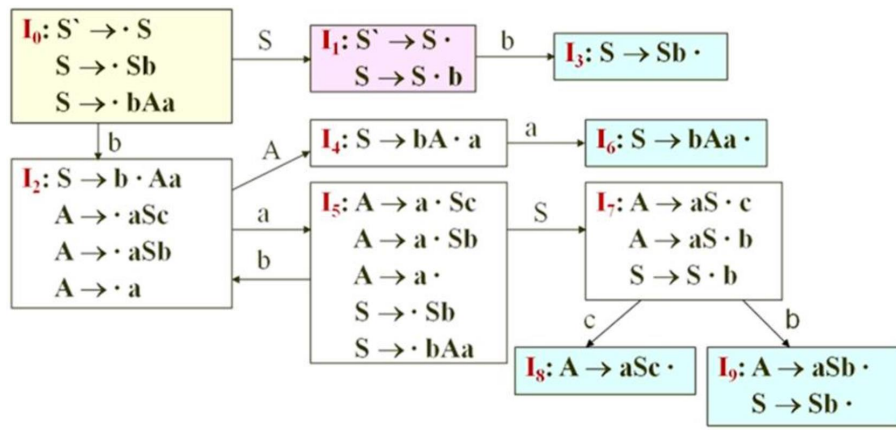
$$S \rightarrow Sb \mid bAa$$
$$A \rightarrow aSc \mid aSb \mid a$$


(3) 判断该文法是否为LR(1)文法，并阐明原因

• 该文法是SLR(1)的，所以自然就是LR(1)文法。

• 理由:

- I1: 由于 $b \notin \text{FOLLOW}(S')$ ，冲突可通过Lookahead解析
- I5: 考虑 $b \notin \text{FOLLOW}(A)$ ，同理可解析
- I9: $\text{FOLLOW}(A) \cap \text{FOLLOW}(S) = \emptyset$



Exercise

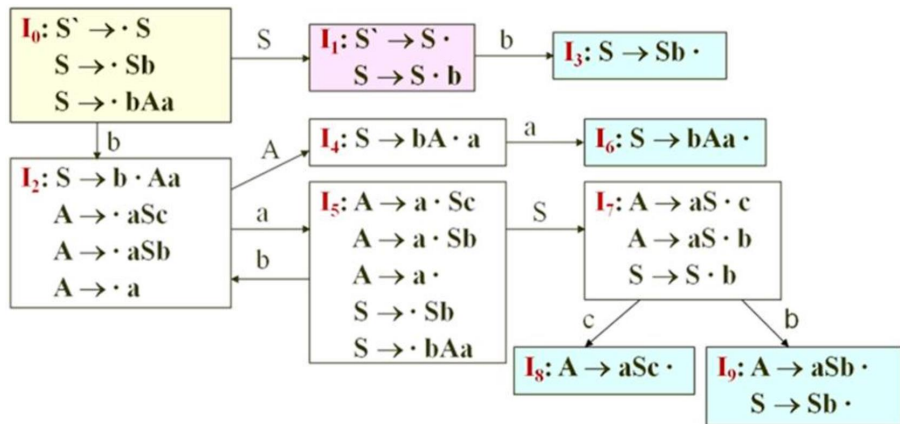


给定文法G(S):

$$S \rightarrow Sb \mid bAa$$

$$A \rightarrow aSc \mid aSb \mid a$$

(4) 试构造该文法的SLR(1)分析表



- (1) $S \rightarrow Sb$
- (2) $S \rightarrow bAa$
- (3) $A \rightarrow aSc$
- (4) $A \rightarrow aSb$
- (5) $A \rightarrow a$

推荐写增广后的文法

| | a | b | c | # | S | A |
|---|------|------|------|------|----|----|
| 0 | | b2 | | | S1 | |
| 1 | | b3 | | OK | | |
| 2 | a5 | | | | | A4 |
| 3 | r(1) | r(1) | r(1) | r(1) | | |
| 4 | a6 | | | | | |
| 5 | r(5) | b2 | | | S7 | |
| 6 | r(2) | r(2) | r(2) | r(2) | | |
| 7 | | b9 | c8 | | | |
| 8 | r(3) | r(3) | r(3) | r(3) | | |
| 9 | r(4) | r(3) | r(3) | r(3) | | |

也可以按照课件
中的分析表写法

Exercise

- 判断题

- 在L-属性定义（L-Attributed Definition）中，每一个属性都是继承属性（Inherited Attribute）。
 - 错。在L-属性定义中，要么是综合属性，要么是受限的继承属性，并非都是继承属性。
- 如果一个翻译模式（Translation Scheme）中存在嵌入在产生式右部的左边或中间的语义动作，则该翻译模式只适合使用递归下降预测翻译器（Recursive Descent Predictive Translator）来实现，无法使用LR分析技术来实现。
 - 错。翻译器分析技术的使用并不是由语义动作的实现决定，而是由文法的特点、语义动作的需求以及性能考虑决定。

Exercise

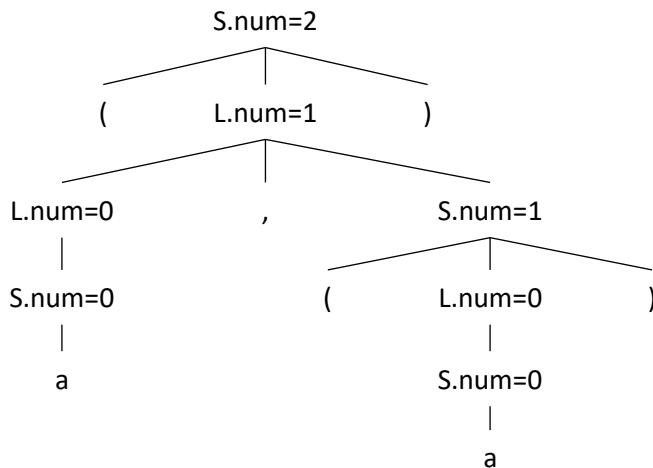
已知文法 $G: S' \rightarrow S$

- $S \rightarrow (L) \mid a$
- $L \rightarrow L, S \mid S$

其语法制导翻译定义如下:

| 产生式 | 语义动作 |
|------------------------|--|
| $S' \rightarrow S$ | $\text{print}(S.\text{num})$ |
| $S \rightarrow (L)$ | $S.\text{num} = L.\text{num} + 1$ |
| $S \rightarrow a$ | $S.\text{num} = 0$ |
| $L \rightarrow L_1, S$ | $L.\text{num} = L_1.\text{num} + S.\text{num}$ |
| $L \rightarrow S$ | $L.\text{num} = S.\text{num}$ |

若输入为 $(a, (a))$ ，且采用自底向上的分析方法，则输出为 2。



Exercise

假设我们有一个产生式 $A \rightarrow BCD$ 。A、B、C、D这四个非终结符号都有两个属性：s是一个综合属性，而i是一个继承属性。对于下面的每组规则，请回答：这些规则是否满足S属性定义的要求？这些规则是否满足L属性定义的要求？并给出理由。

- $A.s = B.s + D.s$
 - 该规则满足S属性定义的要求（每个属性都是综合属性），也满足L属性定义的要求（无继承属性）。
- $A.s = B.i + C.s$
 - 该规则不满足S属性定义的要求（存在继承属性i），满足L属性定义的要求（存在满足限制条件的继承属性i）。
- $A.s = D.i, B.i = A.s + C.s, C.i = B.s, D.i = B.i + C.i$
 - 该规则不满足S属性定义的要求（存在继承属性i），也不满足L属性定义的要求（存在不满足限制条件的继承属性B.i, 其依赖右兄弟的综合属性）。

Exercise

考虑一个非标准的二进制数值系统，其中的每个二进制数 b 的值（value）被定义为从右至左的非零二进制数字表示的十进制数的交替和。例如， $\text{value}(\epsilon) = 0$ ， $\text{value}(10) = 2^1 = 2$ ， $\text{value}(100100) = 2^2 - 2^5 = -28$ ， $\text{value}(11110) = 2^1 - 2^2 + 2^3 - 2^4 = -10$ 。已知表示非标准二进制数字 b 的文法 G 如下：

- $S \rightarrow T \mid \epsilon$
- $T \rightarrow 0T \mid 1T$
- $T \rightarrow 0 \mid 1$

请在表中添加相应的语义动作以计算 $\text{value}(b)$ ，

你只能使用 val 和 tmp 两个属性。

| 产生式 | 语义动作 |
|--------------------------|---|
| $S \rightarrow T$ | $S.\text{val} = T.\text{val};$ |
| $S \rightarrow \epsilon$ | $S.\text{val} = 0;$ |
| $T_1 \rightarrow 0T_2$ | $T_1.\text{tmp} = T_2.\text{tmp} * 2;$ $T_1.\text{val} = T_2.\text{val};$ |
| $T_1 \rightarrow 1T_2$ | $T1.\text{tmp} = T2.\text{tmp} * (-2);$ $T1.\text{val} = T2.\text{val} + T1.\text{tmp};$ |
| $T \rightarrow 0$ | $T.\text{tmp} = -1;$ $T.\text{val} = 0;$ |
| $T \rightarrow 1$ | $T.\text{tmp} = 1;$ $T.\text{val} = 1;$ |