Compilation Principle
编 译 原 理

第13讲：语义分析(3)

张献伟

xianweiz.github.io

DCS290, 04/13/2021

Review Questions (1)

- What is Syntax Directed Translation?

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

- How to augment grammar for semantic analysis?

  Semantic attributes for symbols, rules/actions for productions

- What are SDD and SDT?

  SDD = Syntax Directed Definitions, SDT = SD Translation Schemes

- What are the differences between SDD and SDT?

  SDD = attributes + rules, SDT = attributes + actions.
  SDT is an executable specification of the SDD.

- What is an synthesized attribute?

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

Review Questions (2)

- What is inherited attribute?

  Defined only by attribute values of *N*'s parent, *N* itself and siblings.

- Can a grammar symbol have both *syn* and *inh* attributes?

  Non-terminal: yes; Terminal: only synthesized attributes from lexer.

- What's the usage of dependence graph?

  To decide the evaluation order of attributes.

- Can we always have an evaluation order of the attrs?

  NO. There can be circular dependencies (i.e., cycles in graph).

- What are S-Attributed Definitions (S-SDD)?

  Every attribute is synthesized.

S-Attributed Definitions[S-属性定义]

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

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

L-Attributed Definitions[L-属性定义]

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

Why not synthesized?

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

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

S-SDD or L-SDD?

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

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

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

Not L-SDD: C is right to B

SUN YAT-SEN UNIVERSITY

Syntax Directed Trans. Impl.[实现]

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

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

Syntax Directed Trans. Impl. (cont.)

- Typically, SDT is implemented <u>during parsing</u>[更为高效]
  - Allows compiler to skip parse tree generation
  - Saves time and memory

- Two important classes of SDD's[两个关键子类]
  - SDD is <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 by[SDD->SDT的转换]
  - Placing each action at the end of the production[将每个语义动作都放在产生式的最后]
  - SDTs with all actions at the right ends of the production bodies are called **postfix SDT's** [后缀/尾部SDT]

S-SDD

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

SDT

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

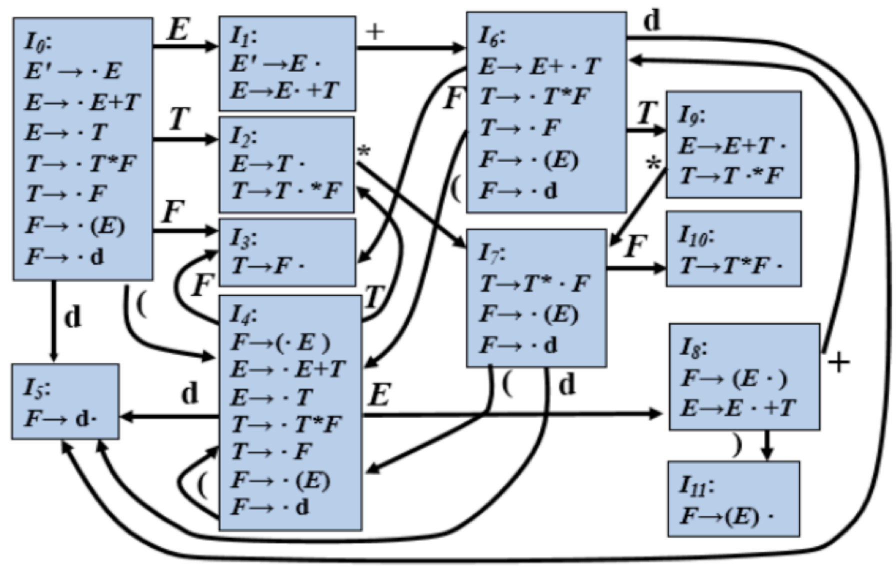
Implement S-SDD (cont.)

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

SDT

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

SLR Automaton



Extend LR Parse Stack[扩展分析栈]

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

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

⇒

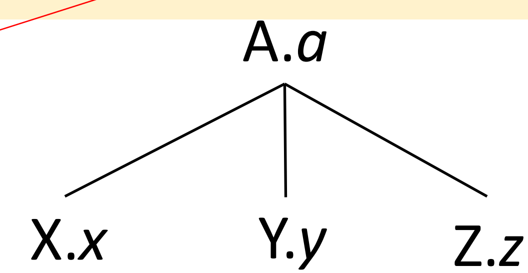
state →	S_0	…	S_n
symbol →	$	…	A
attribute →	-	…	A.*a*
			↑ top

Stack Manipulation[栈操作]

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

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

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



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

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

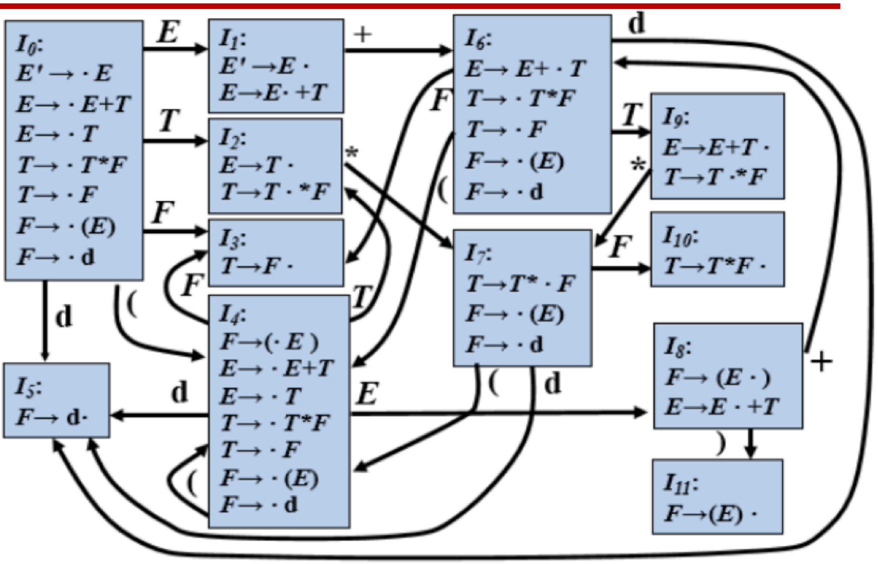
Example

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

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

Example

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

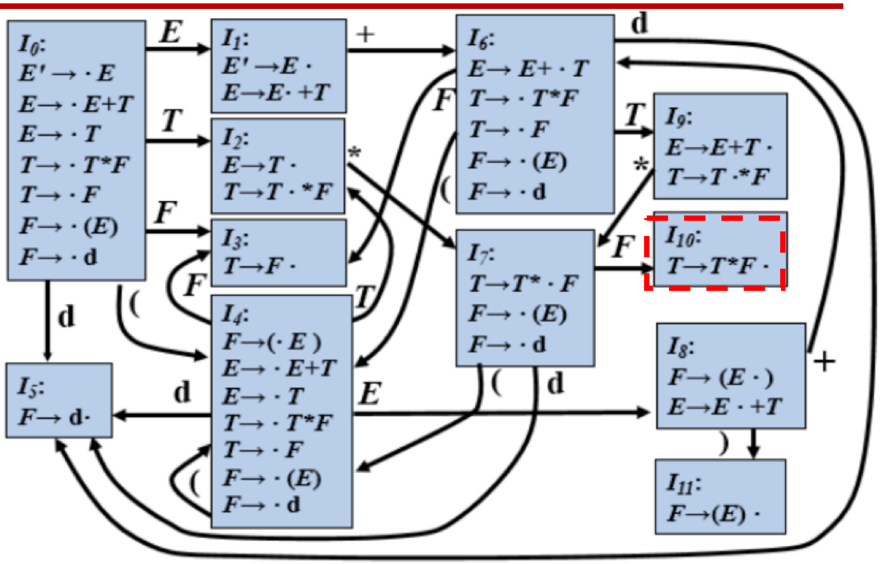


Input: 3 * 5 + 4

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

Example (cont.)

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



Input: 3 * 5 + 4

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

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

== Implement L-SDD ==

- We have examined S-SDD -> SDT
  - S-SDD can be converted to SDT with actions at production ends
  - The SDT can be parsed and translated 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 that occurrence of A in the production body

      [将计算某个非终结符A的继承属性的动作插入到产生式右部中紧靠在A的本次出现之前的位置上]
    - Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production

      将计算一个产生式左部符号的综合属性的动作放在这个产生式右部的末尾]

Example

A -> B C

- C的继承属性：出现之前
- A的综合属性：末尾

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

SDT
(1) T -> F { T’.*inh* = F.*val* } T’ { T.*val* = T’.*syn* }
(2) T’ -> * F { T_1’.*inh* = T’.*inh* x F.*val* } T_1’ { T’.*syn* = T_1’.*syn* }
(3) T’ -> ε { T’.*syn* = T’.*inh* }
(4) F -> digit { F.*val* = digit.*lexval* }

Implement the SDT of L-SDD

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

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

L-SDD in Recursive Decent Parsing

- A recursive-descent parser has a function A for each nonterminal A
  - Non-terminal expansion implemented by a function call
    - (Recursive) calls to functions for non-terminals in RHS

- Synthesized attributes: evaluate at end of function
  - All calls for RHS would have done by then
- Inherited attributes: pass as argument to function
  - Values may come from parent or sibling
  - L-attributed guarantees they have been computed (can only come from already computed portion of RHS)

Example

- Function arguments and return
  - Inherited: arguments
  - Synthesized: return
- Use local variables
- Embed semantic actions

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

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

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

```
T'syn T'(token, T'inh) {
  D: Fval, T1'inh, T1'syn
  if token = "*", then {
     Getnext(token);
     Fval = F(token);
     T1'inh = T'inh x Fval
     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;
}
```

L-SDD in LL Parsing[非递归预测]

- Extend the parse 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 record that represents that terminal[符号位放继承属性]
    - Action-record to evaluate these attributes are immediately above A
  - The synthesized attributes of a nonterminal A are placed in a separate synthesize-record that is immediately below A[综合属性另存放]

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