中山大學 SUN YAT-SEN UNIVERSITY

国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Compilation Principle
编 译 原 理

第15讲：语义分析(3)

张献伟

xianweiz.github.io

DCS290, 4/14/2022

Review Questions

- What's the usage of dependence graph?

  To decide the evaluation order of attributes.

- What is S-SDD?

  Synthesized-SDD, with only synthesized attributes.

- Is the SDD a L-SDD?

A -> X Y Z	Y.*i* = f(Z.*z*, A.*s*)

  NO. Z is right to Y, A.s is synthesized attribute.

- S-SDD is suitable for bottom-up or top-down parsing?

  Bottom-up. Natural to evaluate the parent after seeing all children.

- How to convert an S-SDD into SDT?

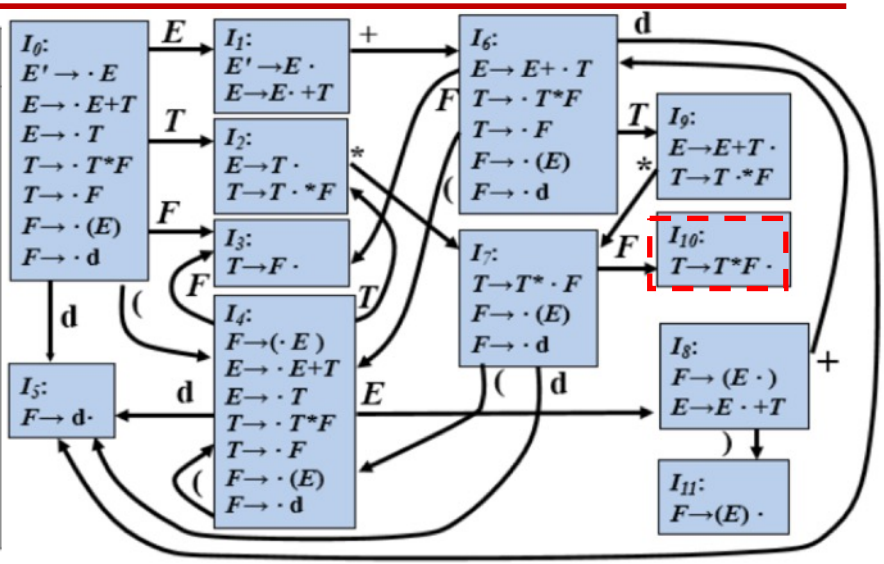
  Place each rule inside '{}' at the end of production.

- For S-SDD in LR-parsing, how to change parse stack?

  Save synthesized attributes into the stack, along with state/symbol.

== Implement S-SDD ==

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

== Implement L-SDD ==

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

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

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

Example

A -> B C

- C的继承属性：出现之前
- A的综合属性：末尾

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

SDT

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

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

(3) T’ -> ε { T’.*syn* = T’.*inh* }

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

Implement the SDT of L-SDD

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

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

L-SDD in Recursive Decent Parsing

- A recursive-descent parser has a function *A* for each nonterminal *A*[递归下降分析方法]
  - 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 sibling
  - L-attributed guarantees they have been computed (can only come from already computed portion of RHS)

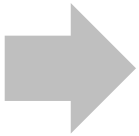
Example

- Function arguments and return[参数和返回值]
  - Arguments: inherited
  - Return: synthesized
- Use local variables[使用局部变量]
- Embed semantic actions[嵌入语义动作]

(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.

L-SDD in LL Parsing (cont.)

- Table-driven LL-parser
  - Mimics a leftmost derivation --> stack expansion

- *A* -> *BC*, suppose nonterminal *C* has an inherited attr *C.i*
  - *C.i* may depend not only on the inherited attr. of A, but on all the attrs of *B*
    - Extra care should be taken on the attribute values
  - Since SDD is L-attributed, surely that the values of the inherited attrs of *A* are available when *A* rises to stack top ($X \rightarrow \alpha A \beta$)
    - Thus, available to be copied into C
  - *A*'s synthesized attrs remain on the stack, below *B* and *C* when expansion happens

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

L-SDD in LL Parsing (cont.)

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

action	Code
A	
A.syn	

Example

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

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

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

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

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

Example (cont.)

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

Input: 3 * 5

Stack top 'digit' matches the input '3'
- pop 'digit', but value copy is needed

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

digit	{ a_6 }	Fsyn	{ a_1 }	T'	T'syn	{ a_2 }	Tsyn	$
lexv=3	d_lexv=3	val =3	val=3	inh	val		val	

完整步骤见👉： MOOC:语法制导翻译-3

L-SDD in LR Parsing[LR解析]

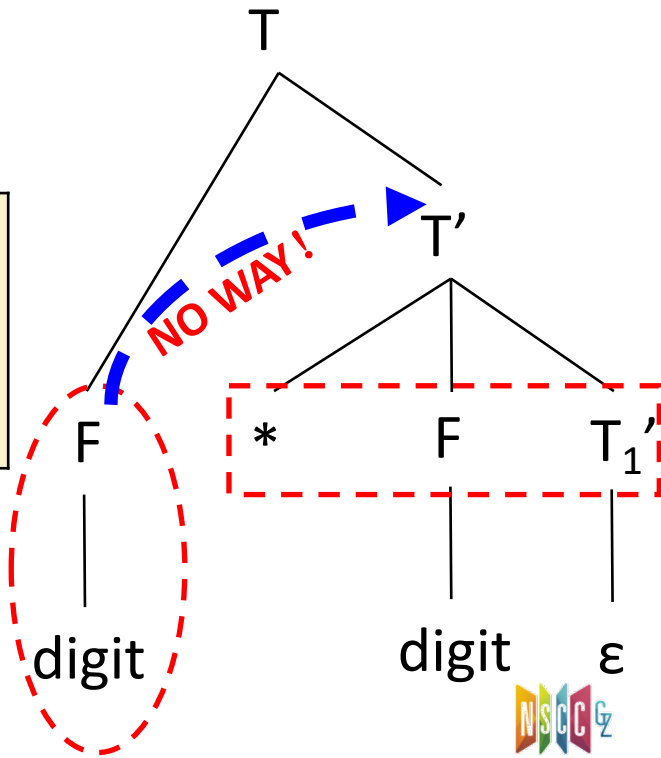
- What we already learnt
  - LR > LL, w.r.t parsing power
    - We can do bottom-up every translation that we can do top-down
  - S-attributed SDD can be implemented in bottom-up way
    - All semantic actions are at the end of productions, i.e., triggered in reduce
- For L-attributed SDD on an LL grammar, can it be implemented during bottom-up parsing?
  - Problem: semantic actions can be in anywhere of the production body

(1) T -> 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* }

The Problem

- It is not natural to evaluate inherited attributes
  - Example: how to get T'.*inh*
- Claim: inherited attributes are on the stack
  - Left attributes guarantee they've already been computed
  - But computed by previous productions – deep in the stack
- Solution
  - **Hack the stack to dig out those values**

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



Marker[标记符号]

- Given the following SDD, where $|\alpha| \neq |\beta|$

  A -> X α { Y.*in* = X.*s* } Y | X β { Y.*in* = X.*s* } Y

  Y -> γ { Y.*s* = f(Y.*in*) }

- Problem: cannot generate stack location for Y.*in*
  - Because X.*s* is at different relative stack locations from Y
- Solution: insert markers M_1, M_2 right before Y

  A -> X α M_1 Y | X β M_2 Y

  Y -> γ { Y.*s* = f(stack[top -$|\gamma|$].*s*) } // Y.*s* = M_1.*s* or Y.*s* = M_2.*s*

  M_1 -> ε { M_1.*s* = stack[top -$|\alpha|$].*s* } // M_1.*s* = X.*s*

  M_2 -> ε { M_2.*s* = stack[top -$|\beta|$].*s* } // M_2.*s* = X.*s*

- **Marker**: a non-terminal marking a location equidistant from the symbol that has an inherited attribute
  - Always produces ε since its only a placeholder for an action

Modify Grammar with Marker[文法修改]

- Given an L-SDD on an LL grammar, we can adapt the grammar to compute the same SDD during a LR parsing
  - Introduce into the grammar a **marker nonterminal**[标记非终结符] in place of each embedded action
    - Each such place gets a distinct marker, and there is one production for any marker *M*, *M* -> ε[空产生式]
  - Modify the action *a* if marker nonterminal *M* replaces it in some production *A* -> α { a } β, and associate with *M* -> ε 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*)
    - Computes attrs in the same way as *a*, but makes those attrs be synthesized attrs of *M* (e,.g., *M.s* = f(*M.i*))

A -> { B.*i* = f(A.*i*); } B C

A -> M B C
M -> ε { M.*i* = A.*i*; M.*s* = f(M.*i*); }

Example

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

(1) T -> F **M** T' { T.*val* = T'.*syn* }
    M -> ε { M.i = F.val; M.s = M.i }
(2) T' -> * F **N** T_1' { T'.*syn* = T_1'.*syn* }
    N -> ε { N.i_1 = T'.inh; N.i_2 = F.val; N.s = N.i_1 x N.i_2 }
(3) T' -> ε { T'.*syn* = T'.*inh* }
(4) F -> digit { F.*val* = digit.*lexval* }

Stack Manipulation[栈操作]

(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* }

(1) T -> F **M** T' { T.*val* = T'.*syn* }
M -> ε { M.i = F.val; M.s = M.i }
(2) T' -> * F **N** T_1' { T'.*syn* = T_1'.*syn* }
N -> ε { N.i1 = T'.inh; N.i2 = F.val; N.s = N.i1 x N.i2 }
(3) T' -> ε { T'.*syn* = T'.*inh* }
(4) F -> digit { F.*val* = digit.*lexval* }

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

完整步骤见👉：MOOC:语法制导翻译-3

Semantic Analysis (3)

Symbol Table

Compilation Phases[编译阶段]

- Lexical analysis[词法分析]
  - Source code → tokens
  - Detects inputs with illegal tokens
  - Is the input program lexically well-formed?
- Syntax analysis[语法分析]
  - Tokens → parse tree or abstract syntax tree (AST)
  - Detects inputs with incorrect structure
  - Is the input program syntactically well-formed?
- Semantic analysis[语义分析]
  - AST → (modified) AST + **symbol table**
  - Detects semantic errors (errors in meaning)
  - Does the input program has a well-defined meaning?

Overview of Symbol Table[符号表]

- **Symbol table** records info of each symbol name in a program[符号表记录每个符号的信息]
  - symbol = name = identifier
- Symbol table is created in the semantic analysis phase[语义分析阶段创建]
  - Because it is not until the semantic analysis phase that enough info is known about a name to describe it
- But, many compilers set up a table at lexical analysis time for the various variables in the program[词法分析阶段准备]
  - And fill in info about the symbol later during semantic analysis when more information about the variable is known
- Symbol table is used in code generation to output assembler directives of the appropriate size and type[后续代码生成阶段使用]

Variable[程序变量]

- What are **variables** in a program?
  - Variables are the names you give to computer memory locations which are used to store values in a computer program
  - Retrieve and update the variables using the names
- Variable **declaration** and **definition**[声明和定义]
  - Declaration: informs the compiler type and name of a variable[类型和名字]
  - Definition: tells the compiler where and how much storage to create for the variable [内存空间分配]

```
// Variable declarations:
extern int x, y;
extern float z;

// Variable definitions:
int x, y;
float z;
```


https://www.tutorialspoint.com/cplusplus/cpp_variable_types.htm

Example

```c
#include <stdio.h>

int g_val;

int main() {
    int l_val;
    static int s_val;

    printf("g_val=%d, l_val=%d, s_val=%d\n", g_val, l_val, s_val);

    return 0;
}
```

```
[xianwei@test>]$ gcc -Wall -g -o testc testc.c
testc.c:9:52: warning: variable 'l_val' is uninitialized when used here [-Wuninitialized]
   printf("g_val=%d, l_val=%d, s_val=%d\n", g_val, l_val, s_val);
                                                   ^~~~~
testc.c:6:13: note: initialize the variable 'l_val' to silence this warning
   int l_val;
            ^
             = 0
1 warning generated.
[xianwei@test>]$ ./testc
g_val=0, l_val=282353718, s_val=0
[xianwei@test>]$ ./testc
g_val=0, l_val=142671926, s_val=0
[xianwei@test>]$ ./testc
g_val=0, l_val=227987510, s_val=0
```

https://en.cppreference.com/w/c/language/storage_class_specifiers