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

Compilation Principle
编 译 原 理

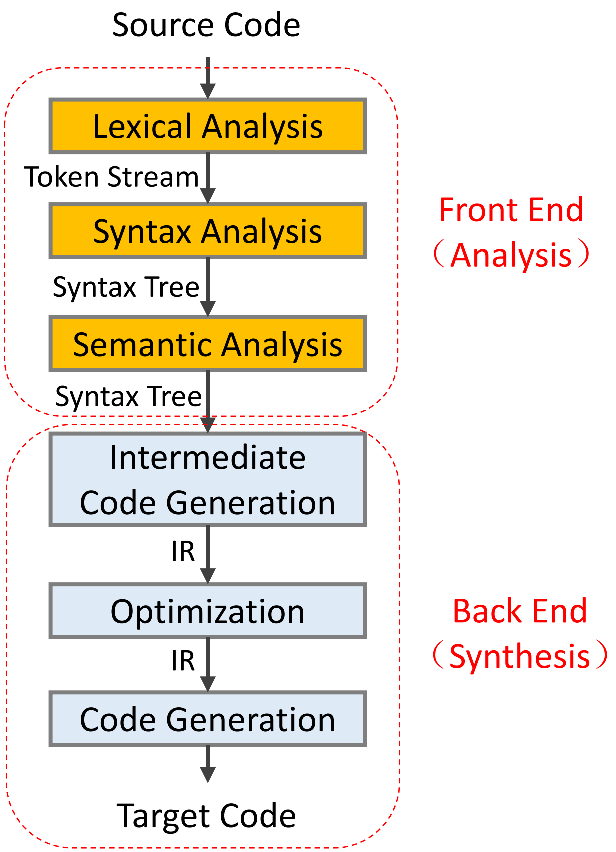
第15讲：语义分析(1)

张献伟

xianweiz.github.io

DCS290, 4/13/2023

Compilation Phases[编译阶段]



Compilation Phases (cont.)

- Lexical analysis[词法分析]
  - Source code → tokens
  - Detects inputs with illegal tokens
  - Is the input program lexically well-formed?

x#y = 1

- 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?

x = 1 y = 2

- 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?

int x; y = x(1)

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 (cont.)

- Semantics of a language is much 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 what the programs means (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 to objects require expressing the pattern:

    {wcw | w ∈ (a|b)*}

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

Semantic Analysis

- Deeper check into the source program[对程序进一步分析]
  - Last stage of the front end[前端的最后阶段]
  - Compiler's last chance to reject incorrect programs[最后拒绝机会]
  - Verify properties that aren't caught in earlier phases
    - Variables are declared before they're used[先声明后使用]
    - Type consistency when using IDs[变量类型一致]
    - Expressions have the right types[表达式类型]
      - E.g., string && bool
    - … …
- 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
  - … …

Example

```
#include <iostream>

using namespace std;

//Derived class
class Child : public Base {
  string myInteger;

  void doSomething() {
    int x[] = {0, 1, 2, 3, 4};
    int z = 'a';
    x[5] = myInteger * y * z;
  }

  void doSomething() {
  }

  int getSum(int n) {
    return doSomething() + n;
  }
};
```

base class not defined

array index out of bounds (runtime)

1) **y variable not declared**

2) cannot multiply a string

cannot redefine functions

cannot add void to int

no main() function

https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/08/Slides08.pdf

Example (cont.)

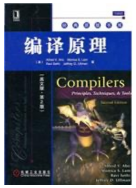
```cpp
#include <iostream>

using namespace std;

//Derived class
class Child : public Base {
  string myInteger;

  void doSomething() {
    int x[] = {0, 1, 2, 3, 4};
    int z = 'a';
    x[5] = myInteger * y * z;
  }

  void doSomething() {
  }

  int getSum(int n) {
    return doSomething() + n;
  }
};
```

```
test.cpp:6:22: error: expected class name
class Child : public Base {
                     ^
test.cpp:15:8: error: class member cannot be redeclared
  void doSomething() {
       ^
test.cpp:9:8: note: previous definition is here
  void doSomething() {
       ^
test.cpp:12:24: error: use of undeclared identifier 'y'
    x[5] = myInteger * y * z;
                       ^
test.cpp:19:26: error: invalid operands to binary expression ('void' and 'int')
    return doSomething() + n;
           ~~~~~~~~~~~~~ ^ ~
4 errors generated.
```

```
test.cpp:6:27: error: expected class-name before '{' token
    6 | class Child : public Base {
      |                           ^
test.cpp:15:8: error: 'void Child::doSomething()' cannot be overloaded with 'void Child::doSomething()'
   15 |   void doSomething() {
      |        ^~~~~~~~~~~
test.cpp:9:8: note: previous declaration 'void Child::doSomething()'
    9 |   void doSomething() {
      |        ^~~~~~~~~~~
test.cpp: In member function 'void Child::doSomething()':
test.cpp:12:24: error: 'y' was not declared in this scope
   12 |     x[5] = myInteger * y * z;
      |                        ^
test.cpp: In member function 'int Child::getSum(int)':
test.cpp:19:26: error: invalid operands of types 'void' and 'int' to binary 'operator+'
   19 |     return doSomething() + n;
      |            ~~~~~~~~~~~~~ ^ ~
      |                       |   |
      |                       void int
```

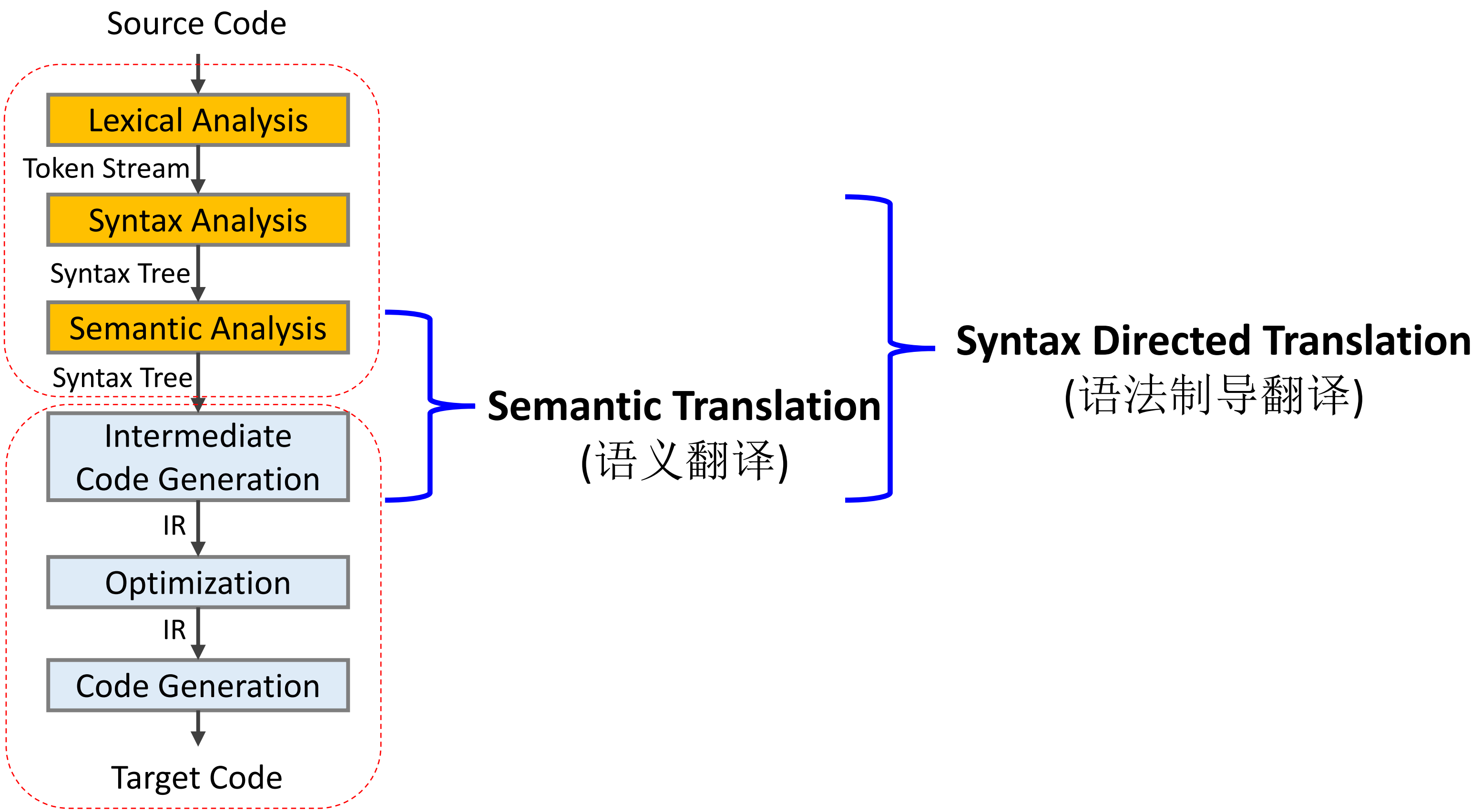
Semantic Analysis: Implementation

- Attribute grammars[属性文法]
  - One-pass compilation
    - Semantic analysis is done right in the middle of parsing
  - Augment rules to do checking during parsing
  - Approach suggested in the Compilers book

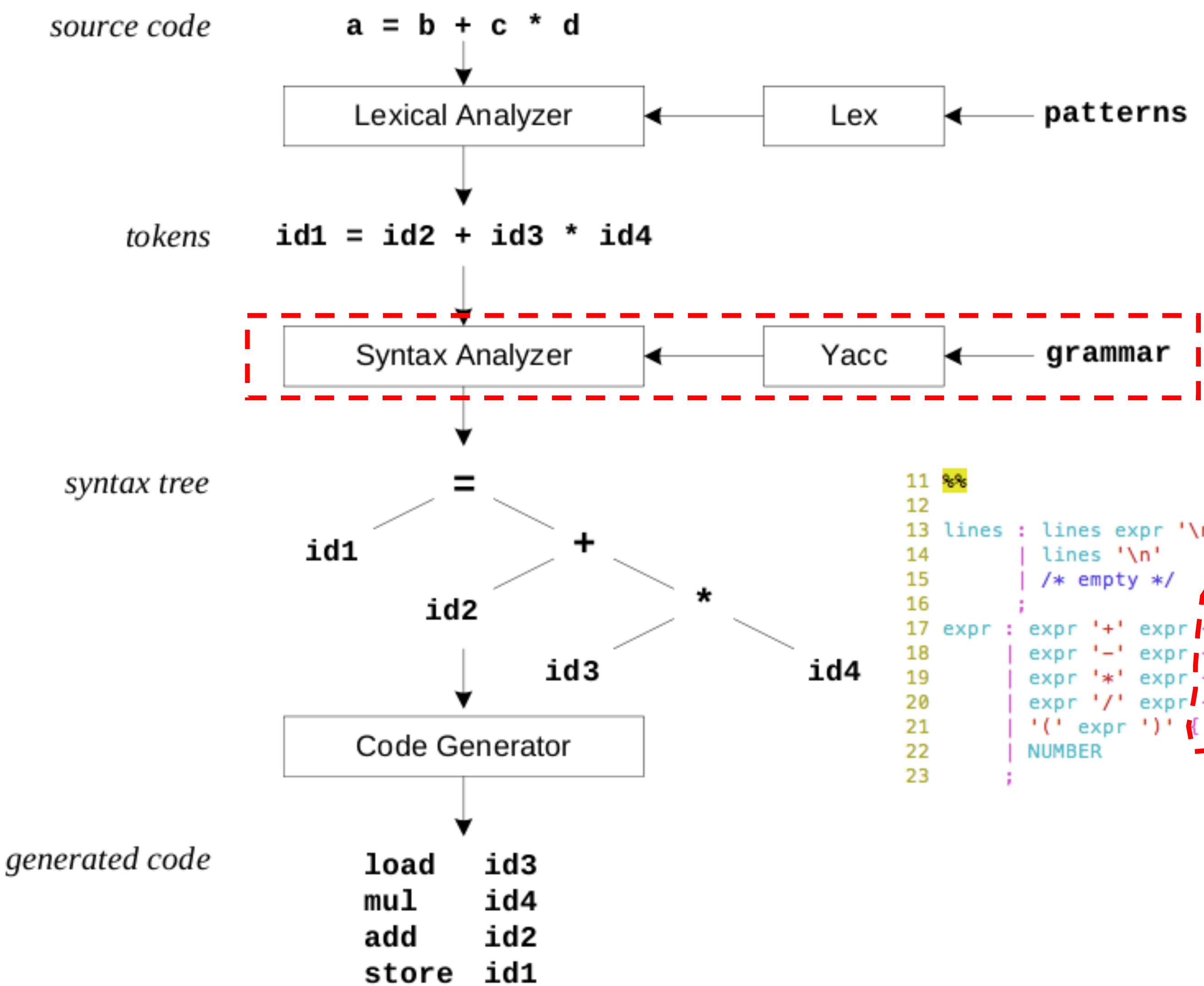


- AST walk[语法树遍历]
  - Two-pass compilation
    - First pass digests the syntax and builds a parse tree
    - The second pass traverses the tree to verify that the program respects all semantic rules
  - Strict phase separation of Syntax Analysis and Semantic Analysis

Syntax Directed Translation[语法制导翻译]



Semantic in Practice

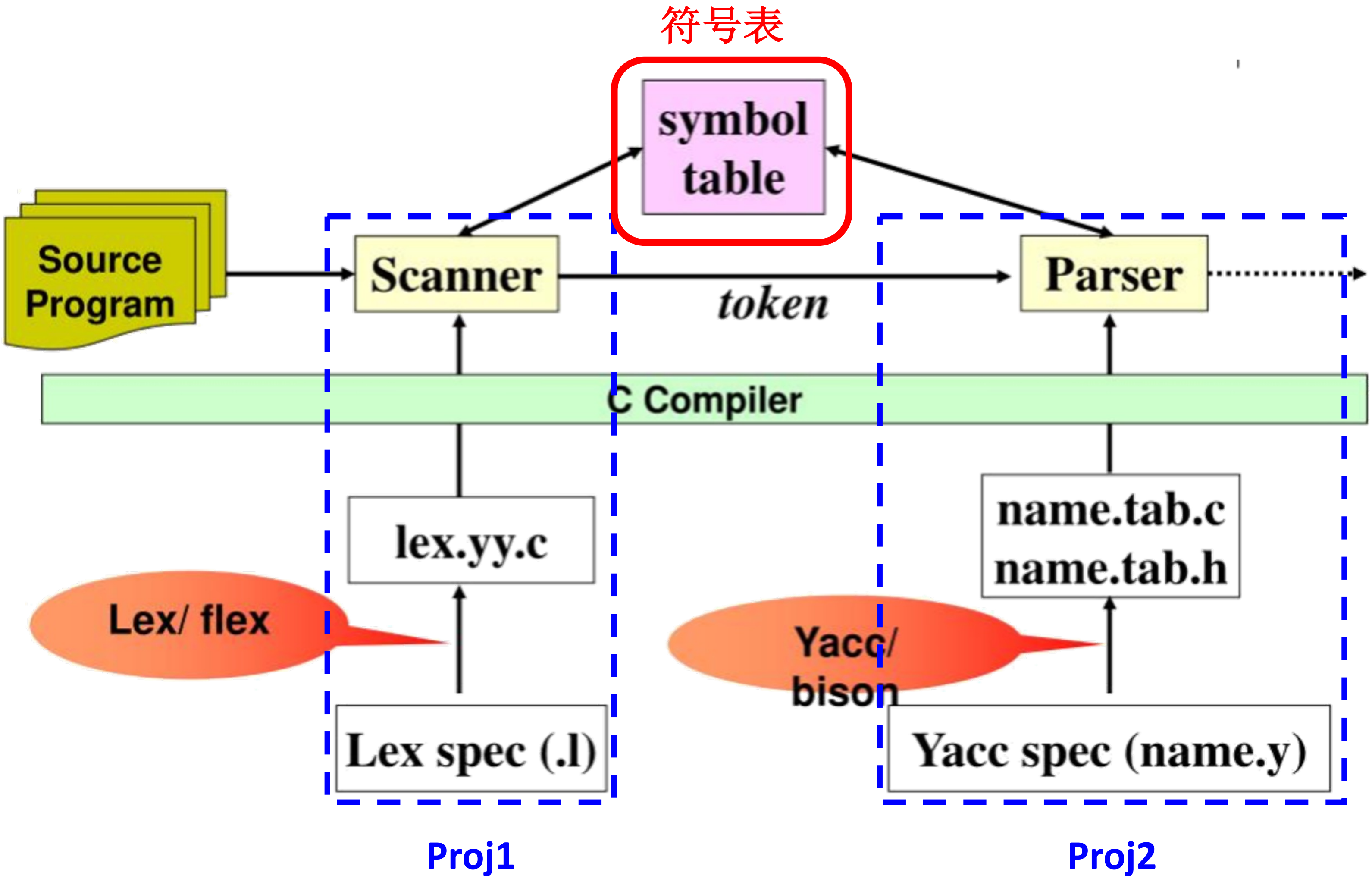


Semantic Actions

```
%%

lines : lines expr '\n' { printf("= %g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
expr : expr '+' expr { $$ = $1 + $3; }
     | expr '-' expr { $$ = $1 - $3; }
     | expr '*' expr { $$ = $1 * $3; }
     | expr '/' expr { $$ = $1 / $3; }
     | '(' expr ')' { $$ = $2; }
     | NUMBER
     ;
```


https://funningboy.blogspot.com/2010/08/lex-yacc-case-study-ply.html

Semantic in Practice (cont.)



https://capsl.udel.edu/courses/cpeg421/2008/slides/lex-yacc_tutorial.pdf

Preview 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]

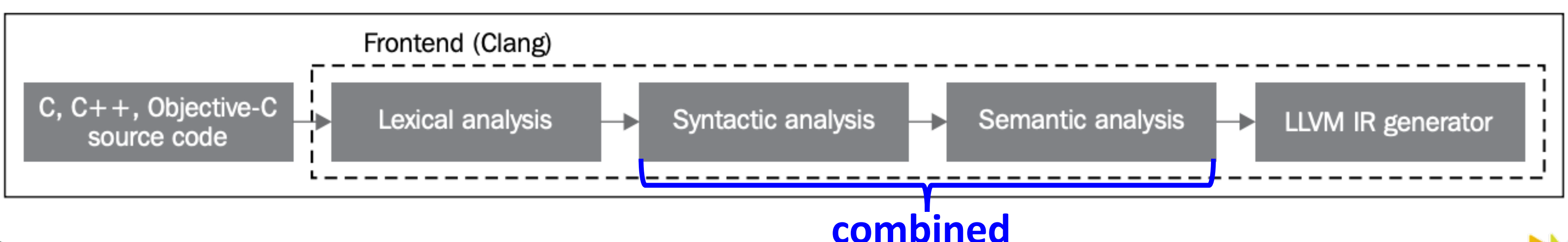
LLVM: Semantic Analysis

- Clang does not traverse the AST after parsing
  - Instead, it performs type checking on the fly, together with AST node generation

```
StmtResult Parser::ParseIfStatement(SourceLocation *TrailingElseLoc) {

  // perform semantic checking for the if statement, emitting diagnostics accordingly
  return Actions.ActOnIfStmt(IfLoc, IsConstexpr, InitStmt.get(), Cond,
                             ThenStmt.get(), ElseLoc, ElseStmt.get());
}
```

https://github.com/llvm-mirror/clang/blob/master/lib/Parse/ParseStmt.cpp
https://clang.llvm.org/doxygen/ParseAST_8cpp_source.html

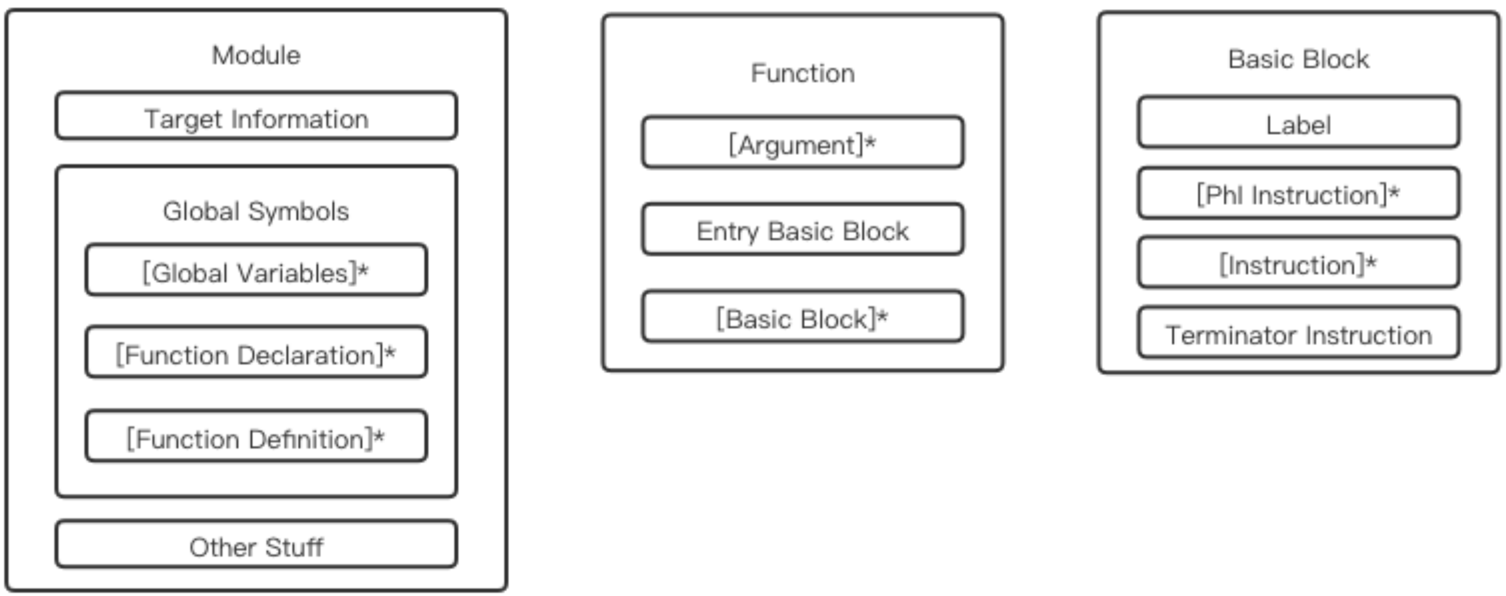
  - After the combined parsing and semantic analysis, the ParseAST function invokes the method HandleTranslationUnit to trigger any client that is interested in consuming the final AST.



Bruno Cardoso Lopes and Rafael Auler, Getting Started with LLVM Core Libraries

LLVM: Module

- The **Module** class represents the top level structure present in LLVM programs
  - An LLVM module is effectively either a translation unit of the original program or a combination of several translation units merged by the linker
  - The Module class keeps track of a list of Functions, a list of GlobalVariables, and a **SymbolTable**



https://releases.llvm.org/7.1.0/docs/ProgrammersManual.html#symboltable

LLVM: Symbol Table

- Public members of Module class
  - *SymbolTable *getSymbolTable()*
    - Return a reference to the SymbolTable for this Module
  - *Function *getOrInsertFunction(const std::string &Name, const FunctionType *T)*
    - Look up the specified function in the Module SymbolTable. If it does not exist, add an external declaration for the function and return it.
  - *std::string getTypeName(const Type *Ty)*
    - If there is at least one entry in the SymbolTable for the specified Type, return it. Otherwise return the empty string
  - *bool addTypeName(const std::string &Name, const Type *Ty)*
    - Insert an entry in the SymbolTable mapping Name to Ty. If there is already an entry for this name, true is returned and the SymbolTable is not modified.

https://llvm.org/doxygen/classllvm_1_1Module.html

Syntax Directed Translation[语法制导翻译]

- To translate based on the program's syntactic structure[语法结构]
  - Syntactic structure: structure of a program given by grammar
  - 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:
  - Attach **semantic attributes**[语义属性] to each grammar symbol
    - The attributes describe the symbol properties
    - An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register …
  - For each grammar production, give **semantic rules or actions**[语义规则或动作]
    - The actions describe how to compute the attribute values associated with each symbol in a production

Attributes[语义属性]

- Attributes can represent anything depending on the task[属性可以表示任意含义]
  - If computing expression: *a number (value of expression)*
  - If building AST: *a pointer (pointer to AST for expression)*
  - If generating code: *a string (assembly code for expression)*
  - If type checking: *a type (type for expression)*
- Format: *X.a* (*X* is a symbol, *a* is one of its attributes)
- For Project 2 – Syntax Analysis
  - Semantic attributes
    - *Name, type*
  - Semantic actions

```
%%

lines : lines expr '\n' { printf("= %g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
expr : expr '+' expr { $$ = $1 + $3; }
     | expr '-' expr { $$ = $1 - $3; }
     | expr '*' expr { $$ = $1 * $3; }
     | expr '/' expr { $$ = $1 / $3; }
     | '(' expr ')' { $$ = $2; }
     | NUMBER
     ;
```

```
CompUnit: FuncDef {
  auto inner = stak.back();
  stak.pop_back();
  stak.push_back(llvm::json::Object{{"kind", "TranslationUnitDecl"},
                                    {"inner", llvm::json::Array{inner}}});
}
```

How to Specify Syntax Directed Translation

- **Syntax Directed Definitions** (SDD)[语法制导定义]
  - Attributes + semantic rules[语义规则]for computing them
    - 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]
  - Example rules for computing the value of an expression

    $E \text{ -> } E_1 + E_2$ RULE: $\{E.val = E_1.val + E_2.val\}$

    $E \text{ -> } id$ RULE: $\{E.val = id.lexval\}$

- **Syntax Directed Translation scheme** (SDT)[语法制导翻译方案]
  - Attributes + semantic actions[语义动作] for computing them
  - Example actions for computing the value of an expression

    $E \text{ -> } E_1 + E_2$ $\{E.val = E_1.val + E_2.val\}$

    $E \text{ -> } id$ $\{E.val = id.lexval\}$

SDD vs. SDT

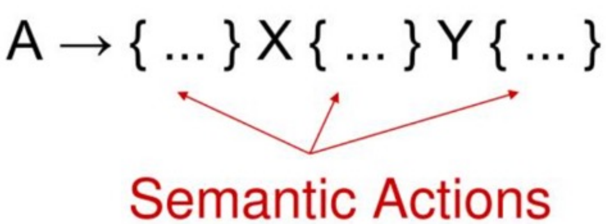
- SDD[语法制导定义]: 是CFG的推广，翻译的高层次规则说明
  - A CFG grammar together with attributes and semantic rules
    - A subset of them are also called **attribute grammars**[属性文法]
      - No side effects, i.e., rules are strictly local to each production
  - Semantic rules imply no order to attribute evaluation
- SDT[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

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

SDD vs. SDT (cont.)

- Syntax: A -> α {*action*$_1$} β {*action*$_2$} *γ* …
- 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 vs. 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



SDD[语法制导定义]

- SDD has two types of attributes[两种属性]
  - For a non-terminal *A* at a parse-tree node *N*

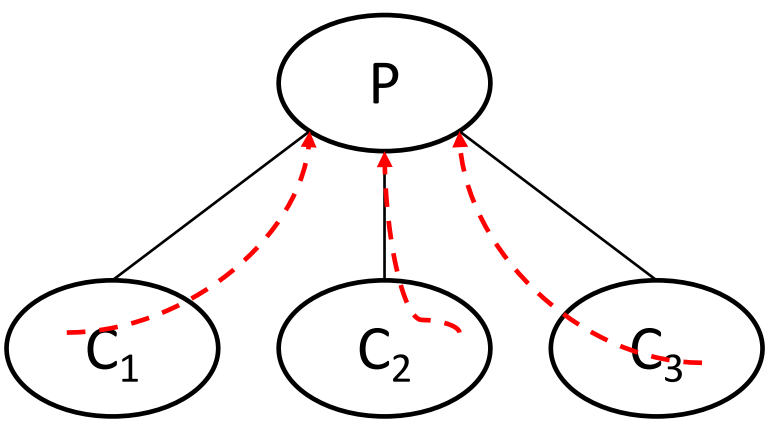
- **Synthesized attribute**[综合属性]
  - Defined by a semantic rule associated with the production at *N*
    - The production must have *A* as its head (i.e., A -> …)
  - A synthesized attribute of node *N* is defined only by attribute values at *N*'s children and *N* itself[子节点或自身]
- **Inherited attribute**[继承属性]
  - Defined by a semantic rule associated with the production at the parent of *N*
    - The production must have *A* as a symbol in its body (i.e., … -> …A…)
  - An inherited attributed at 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]

中山大學 SUN YAT-SEN UNIVERSITY

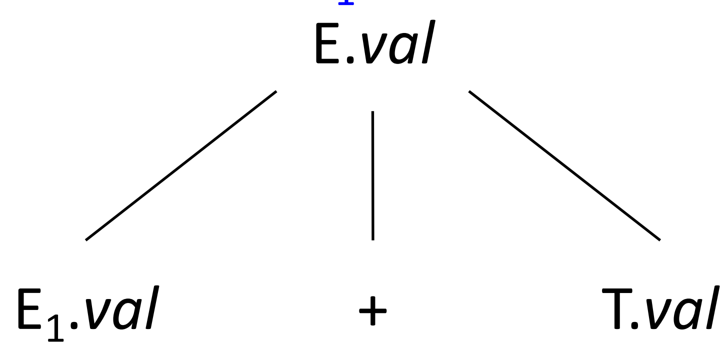
NSCC GZ

Synthesized Attribute[综合属性]

- Synthesized attribute for non-terminal *A* of parse-tree node *N*[非终结符的综合属性]
  - Only defined by *N*'s children and *N* itself
    - Passed up the tree
  - P.syn_attr = f(P.attrs, C_1.attrs, C_2.attrs, C_3.attrs)
- Terminals can have synthesized attributes[终结符综合属性]
  - Lexical values supplied by the lexical analysis
  - Thus, no semantic rules in SDD for terminals

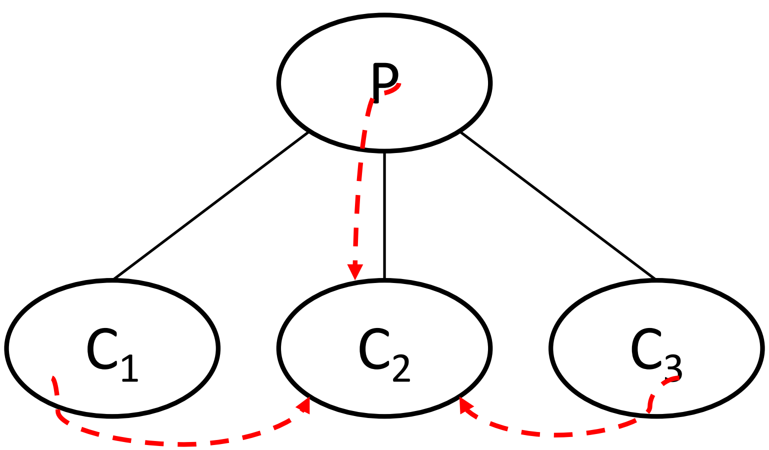


$E.val = E_1.val + T.val$

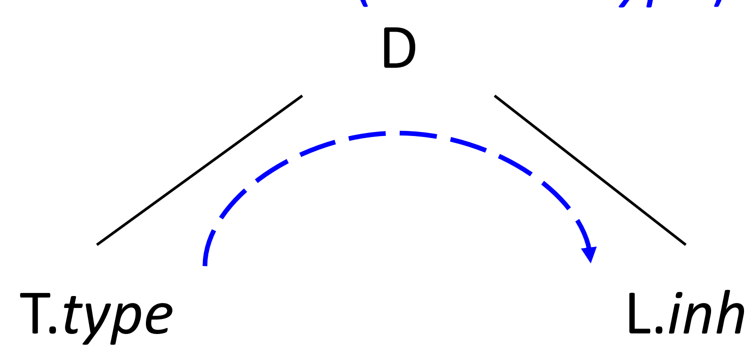


Inherited Attribute[继承属性]

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

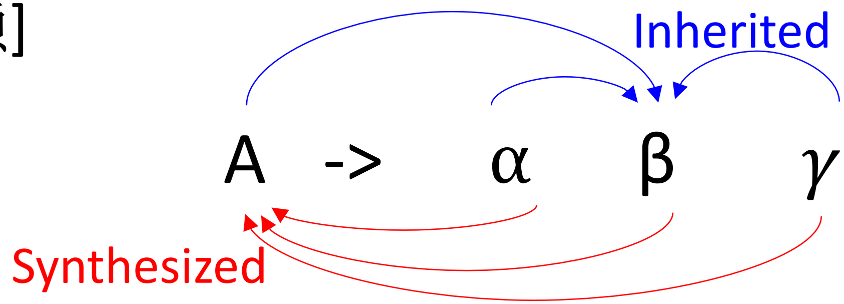


D -> T L (L.*inh* = T.*type*)



SDD[语法制导定义]

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



- SDD has rule of the form for each grammar production

  b = f(A.attrs, α.attrs, β.attrs, γ.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[综合]

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*

Each non-terminal has a single synthesized attribute val
Terminal *digit* has a synthesized attribute lexval

Arithmetic expressions with + and *

(1) Print the numerical value of the entire expression
(2) Compute value of summation
(3) Value copy
(4) Compute value of multiplication
(5) Value copy
(6) Value copy

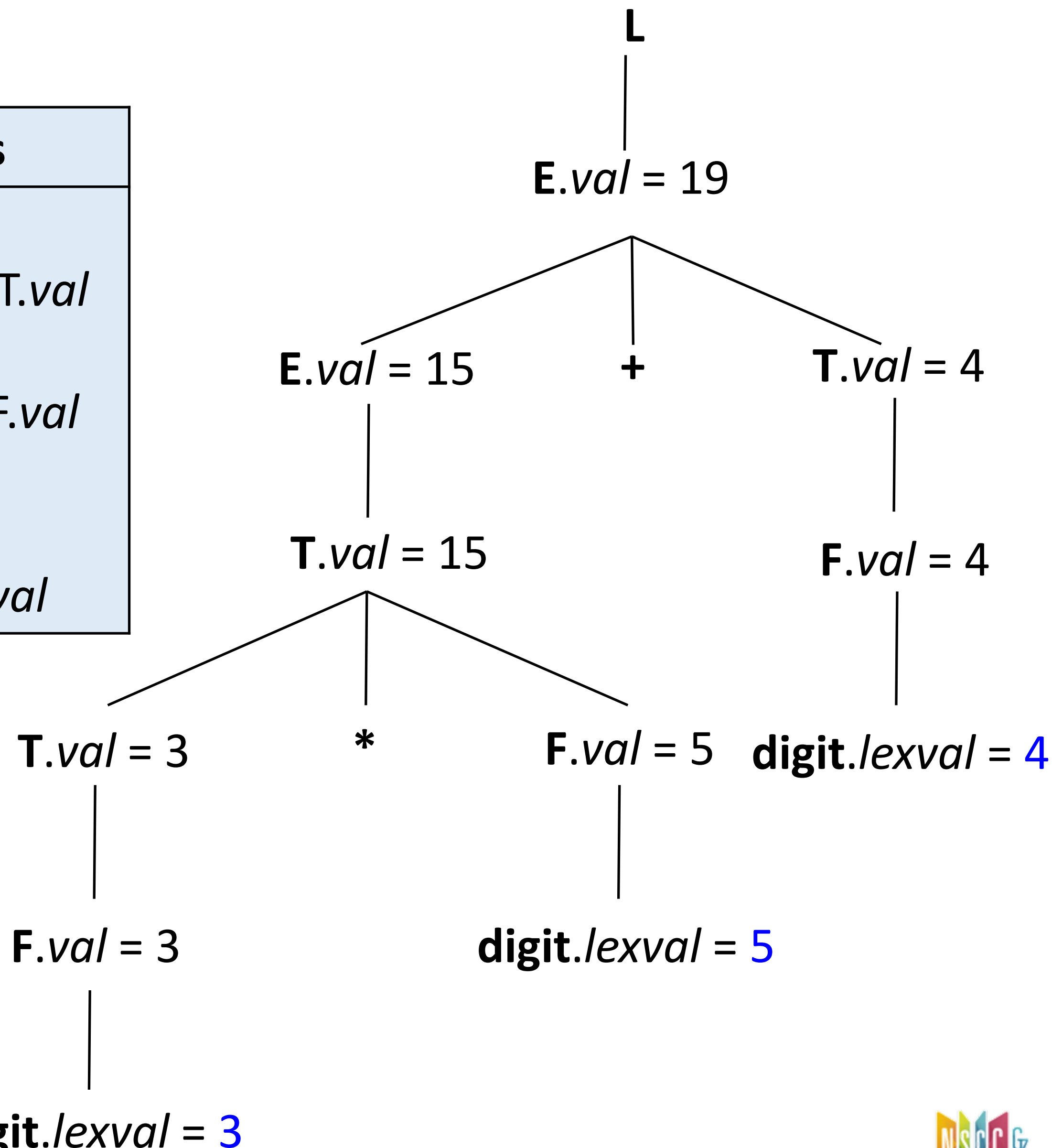
Example: Synthesized Attribute (cont.)

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*

Input:

3 * 5 + 4



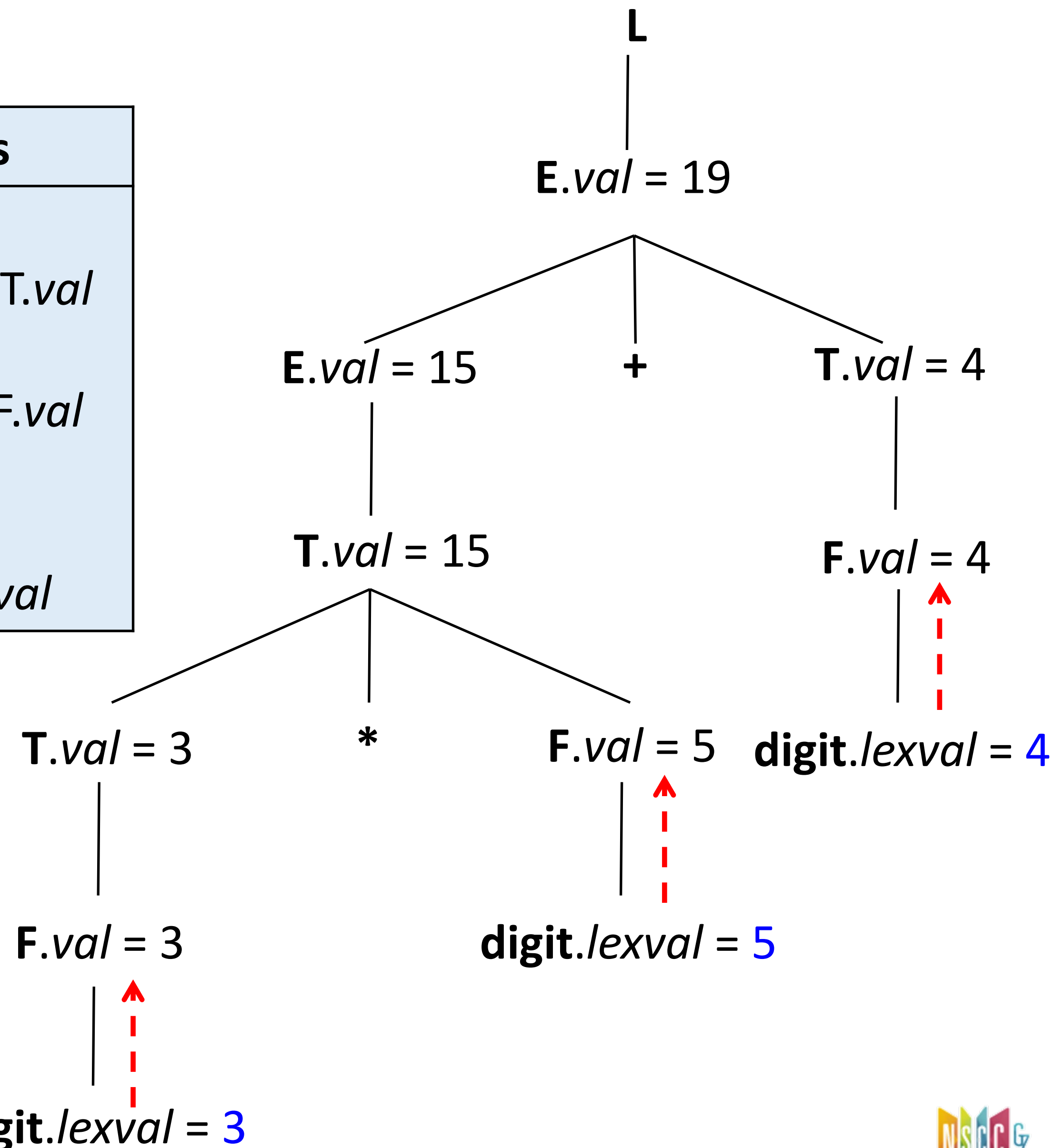
Example: Synthesized Attribute (cont.)

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*

Input:

3 * 5 + 4



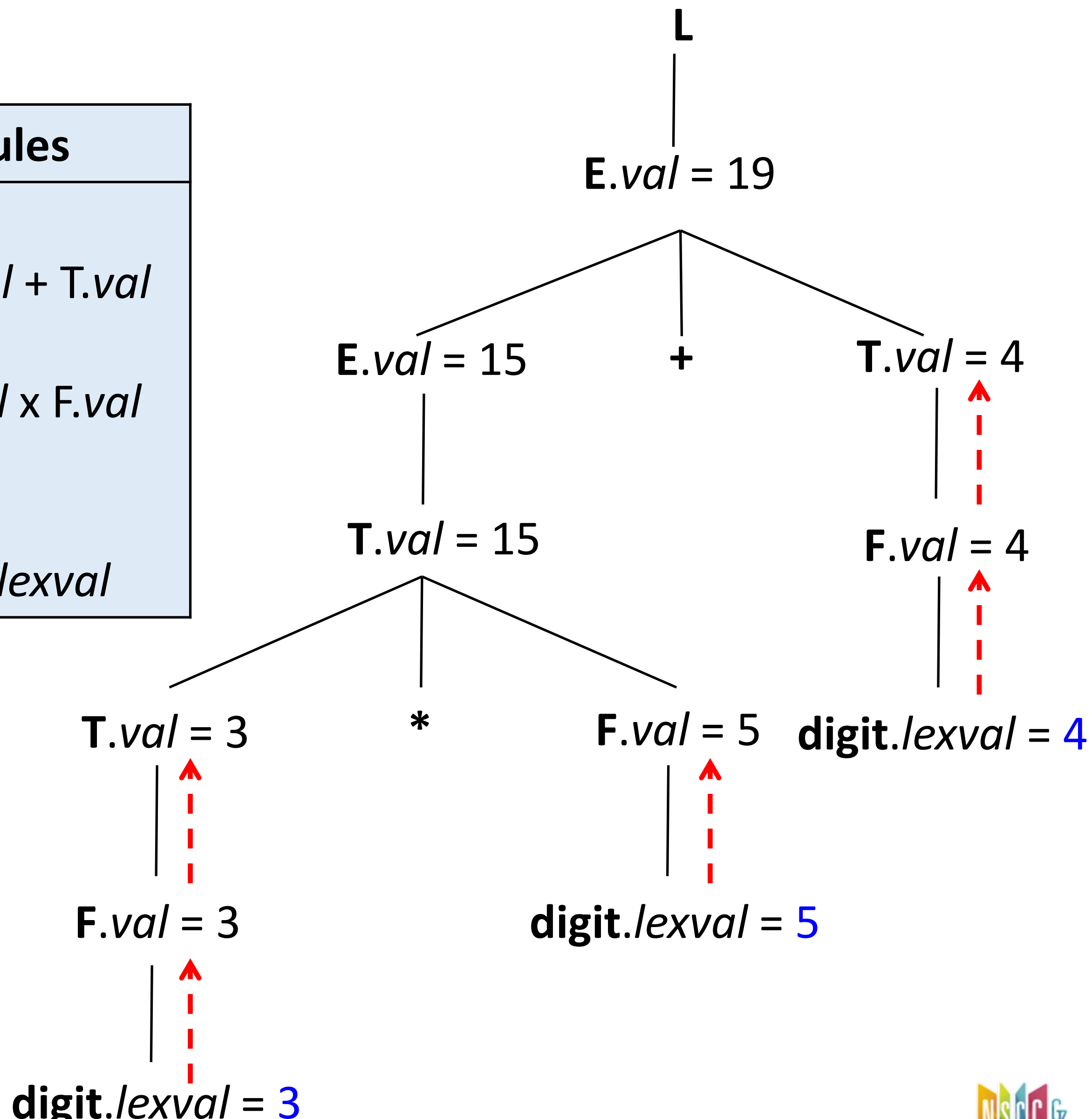
Example: Synthesized Attribute (cont.)

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*

Input:

3 * 5 + 4



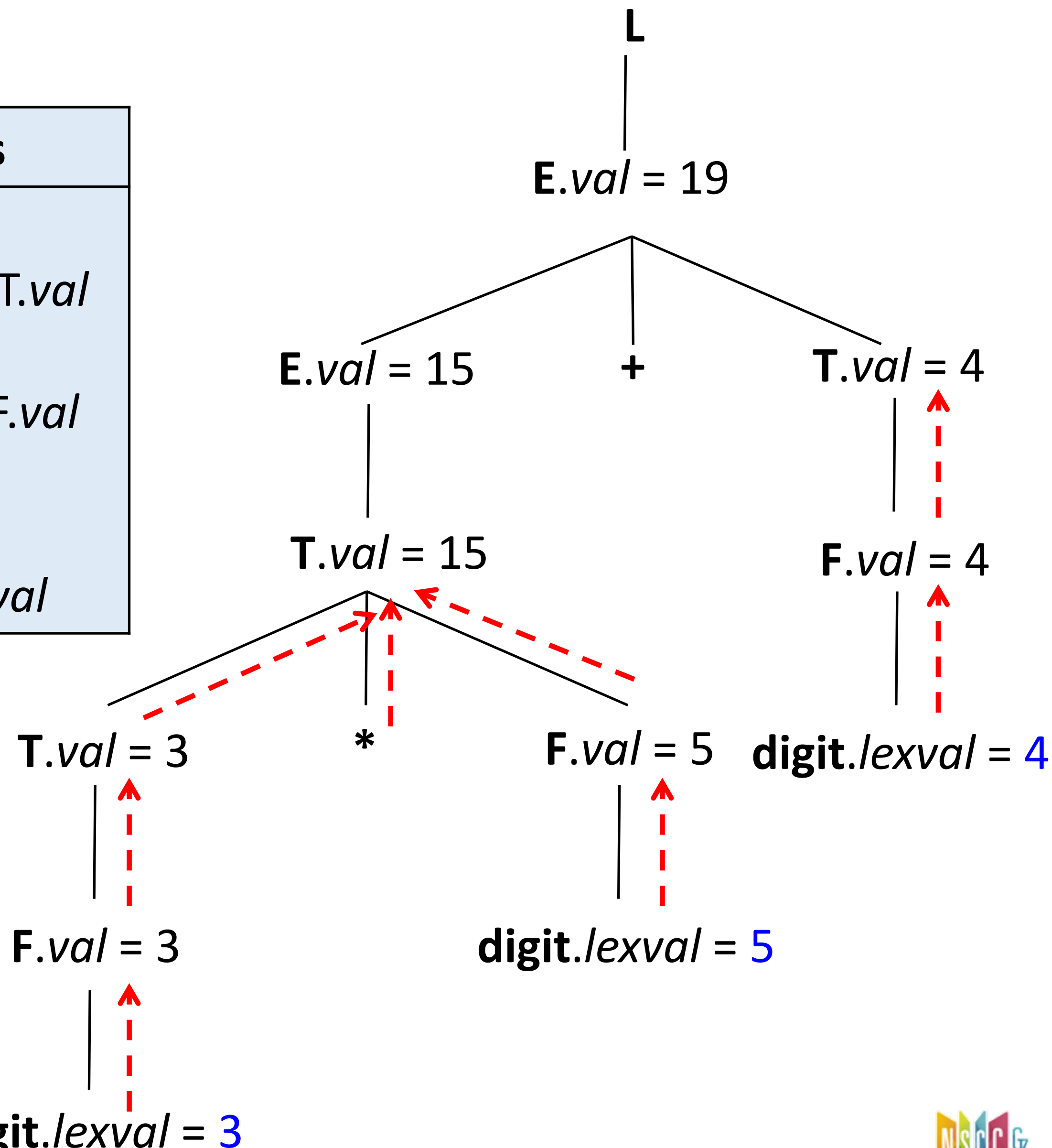
Example: Synthesized Attribute (cont.)

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*

Input:

3 * 5 + 4



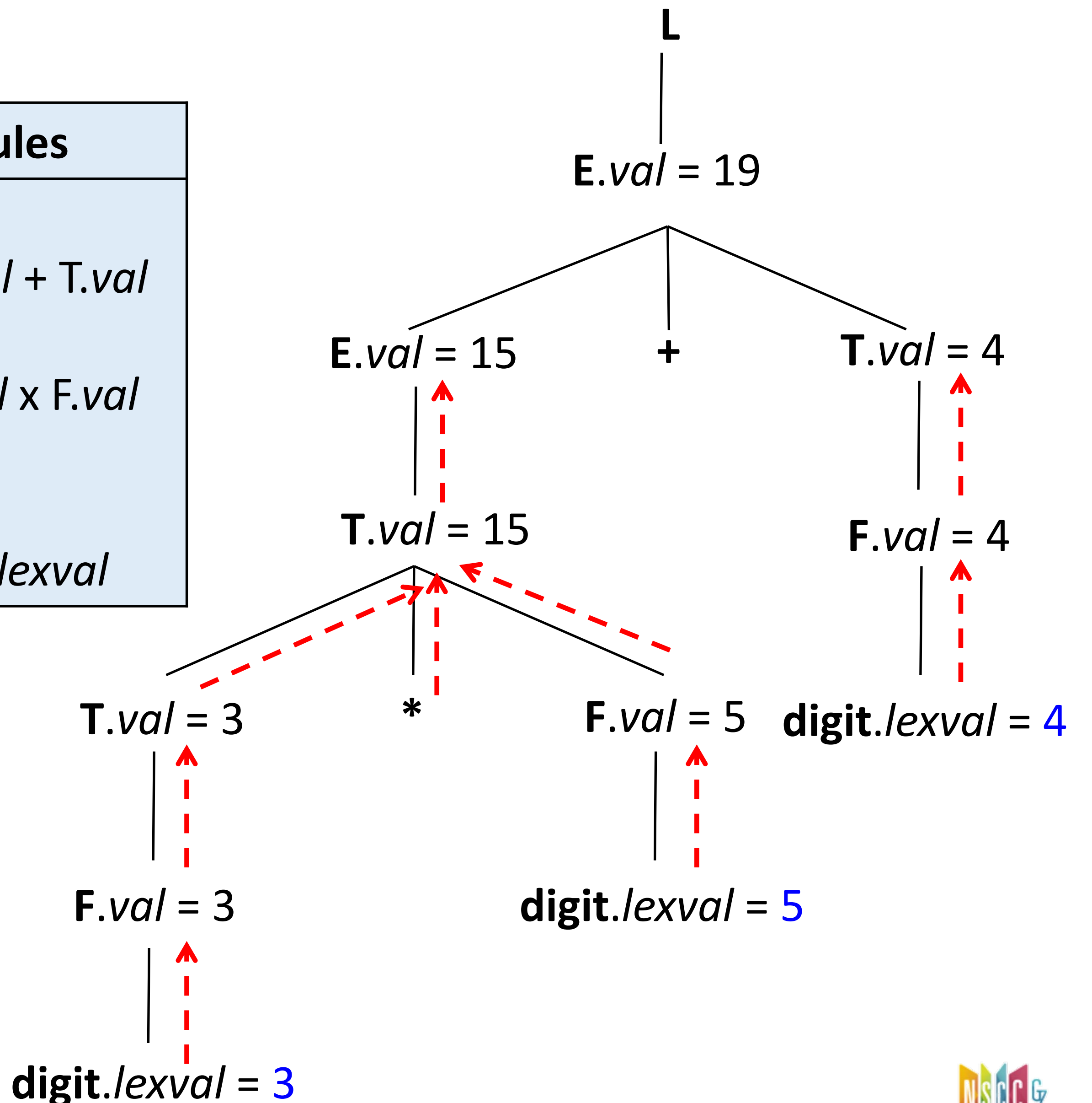
Example: Synthesized Attribute (cont.)

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*

Input:

3 * 5 + 4



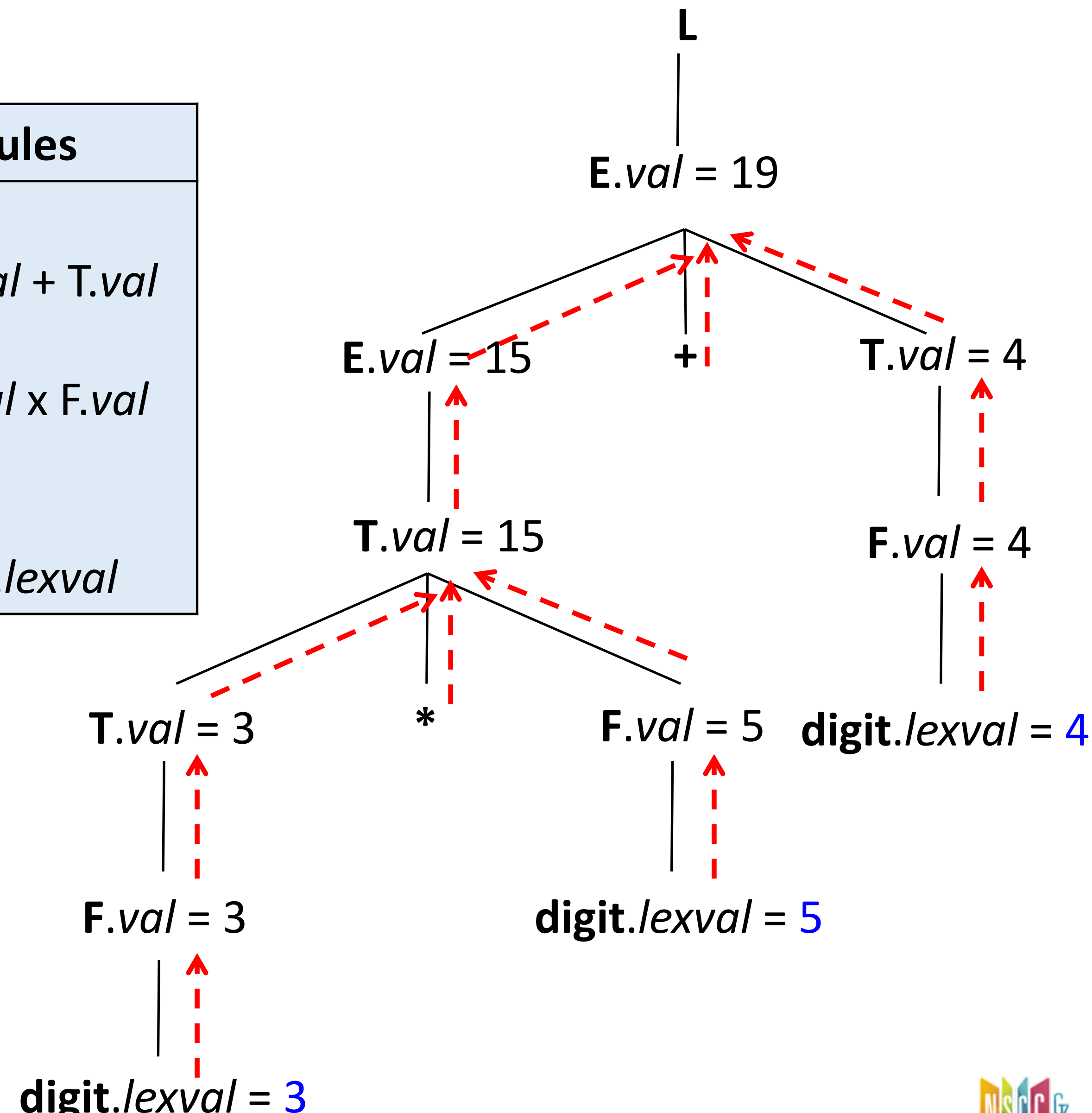
Example: Synthesized Attribute (cont.)

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*

Input:

3 * 5 + 4



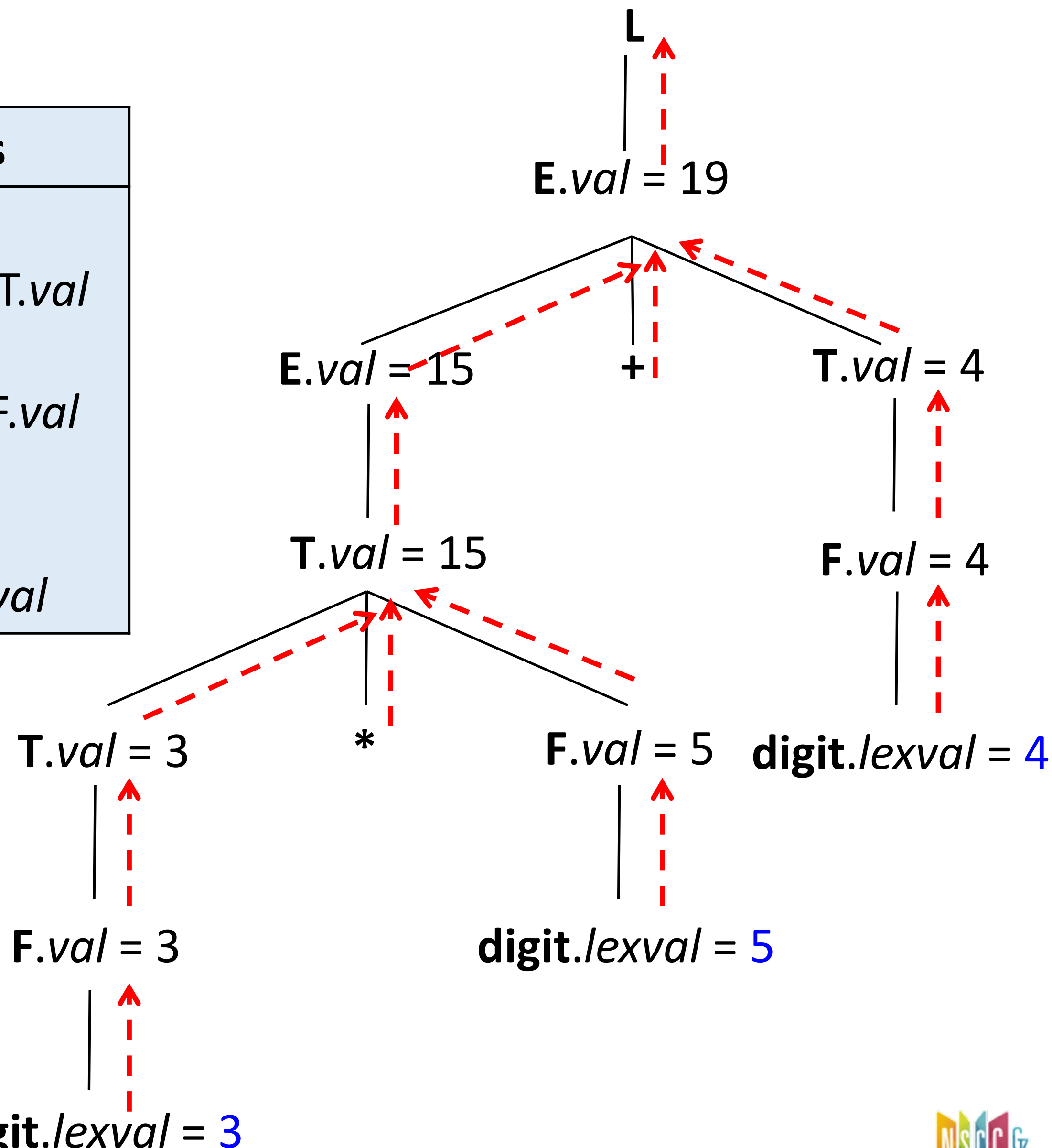
Example: Synthesized Attribute (cont.)

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*

Input:

3 * 5 + 4



Example: Synthesized Attribute (cont.)

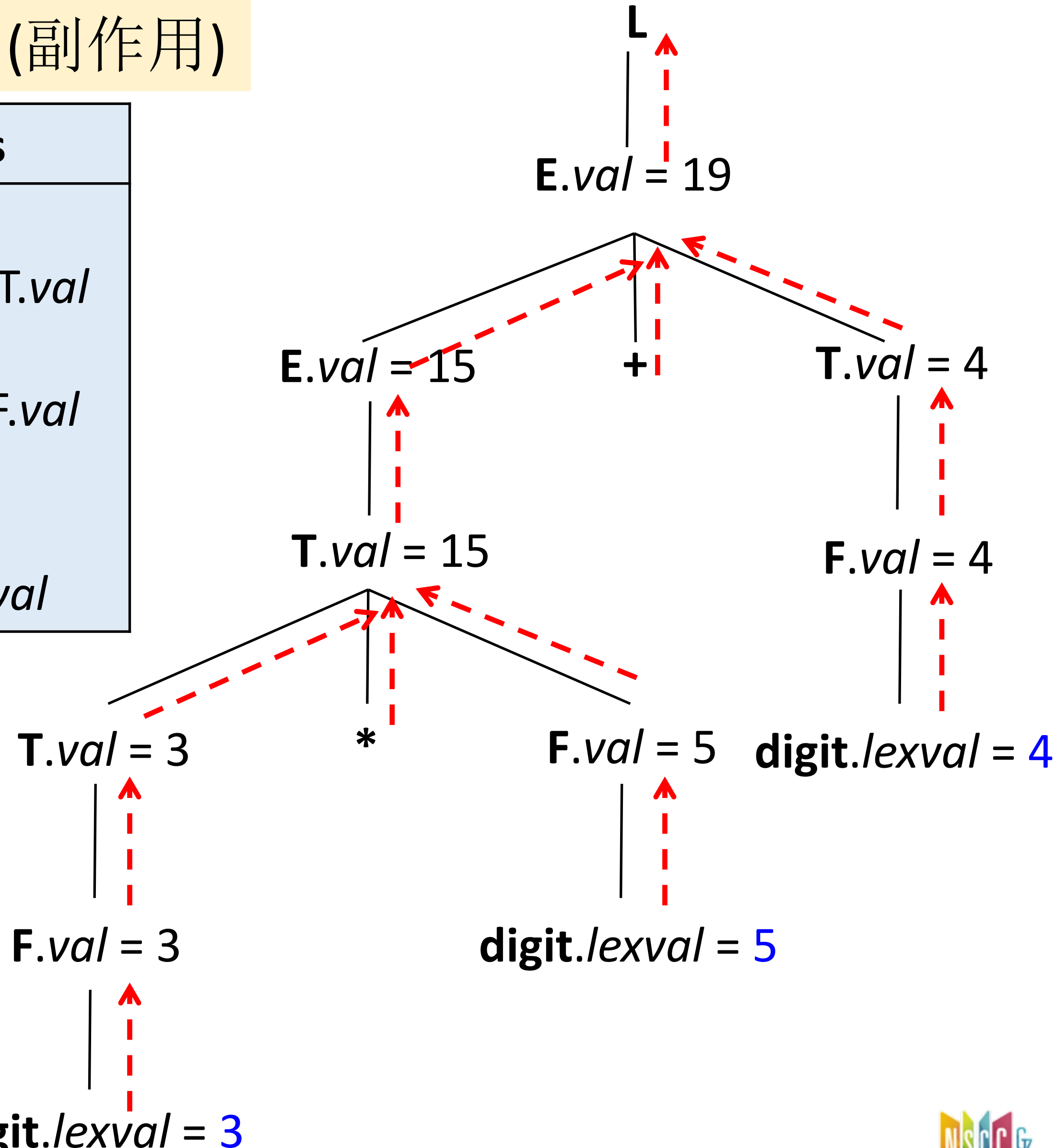
SDD:

Side effect (副作用)

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

Input:

3 * 5 + 4



Example: Synthesized Attribute (cont.)

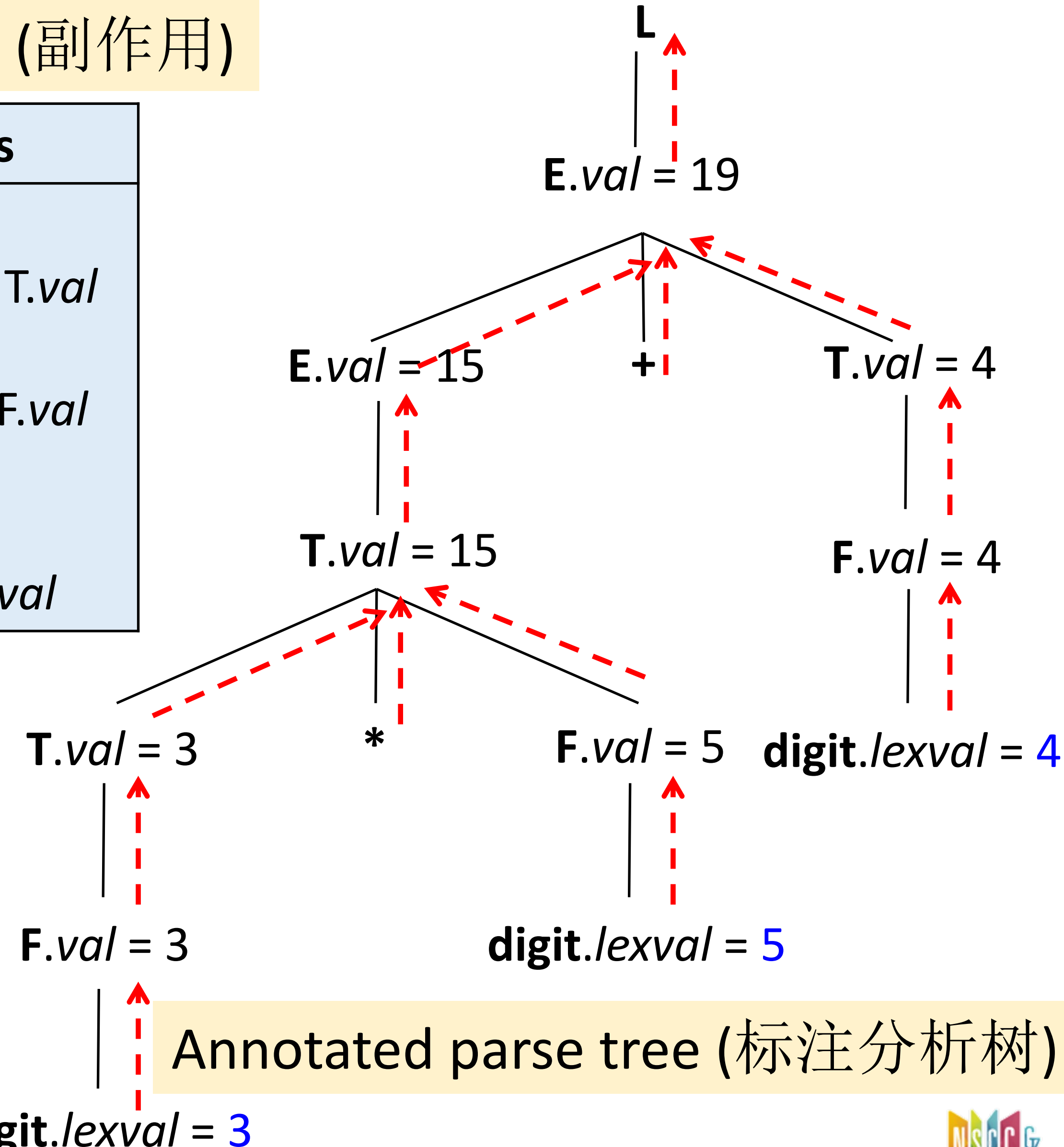
SDD:

Side effect (副作用)

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*

Input:

3 * 5 + 4



Annotated parse tree (标注分析树)