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, 4/23/2024

Review Questions

- How does LR(1) improve SLR(1)?

  Use exact lookaheads, instead of the FOLLOW, to reduce; split states

- Drawbacks of LR(1)?

  More states, i.e., more rows, i.e., higher storage overhead

- How does LALR relate to SLR(1) and LR(1)?

  Compromise LR(1) and SLR(1), keep LR(1) parse power and SLR(1) table size

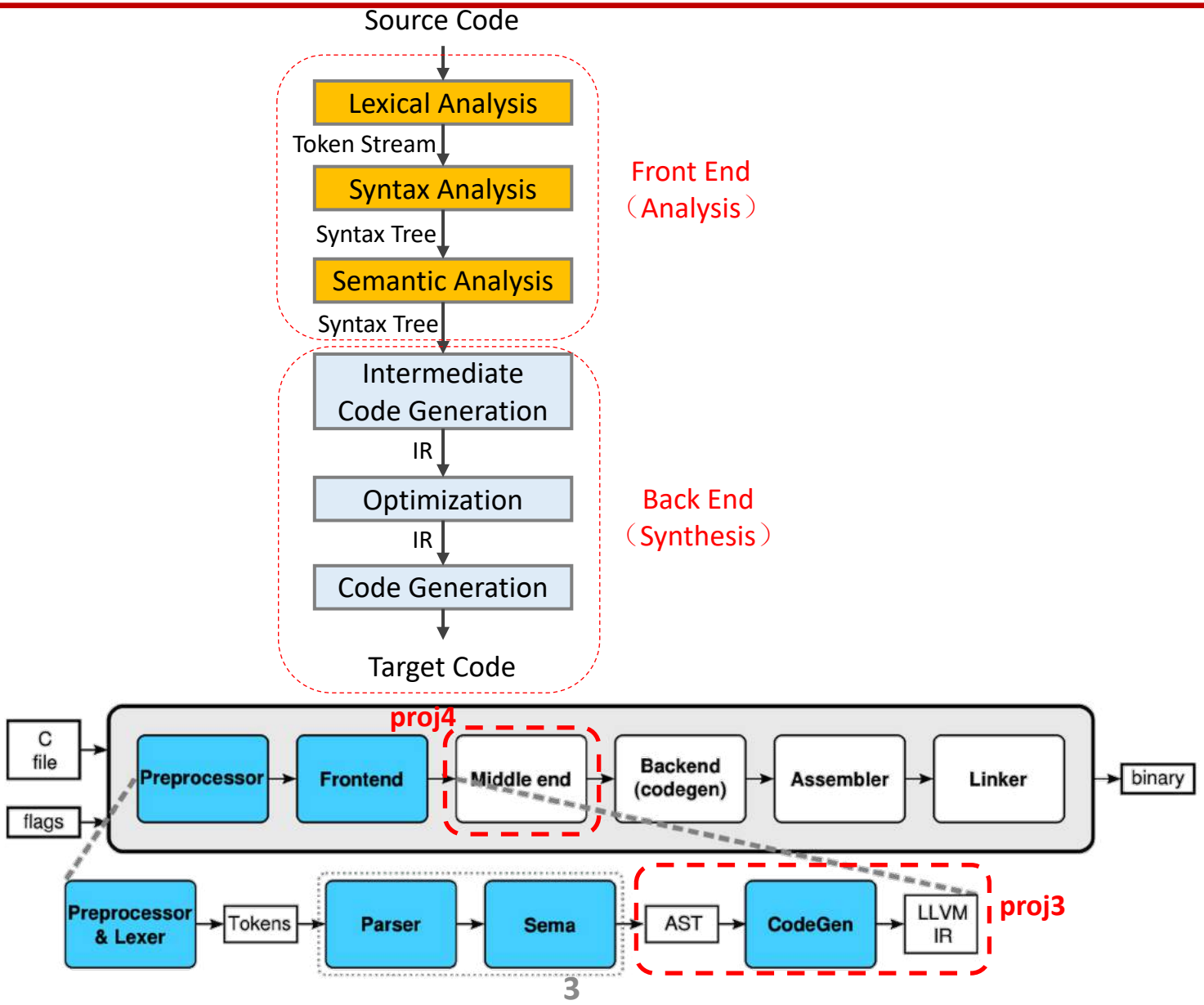
- Can LALR introduce new shift-reduce conflicts?

  No. Merging can only introduce new reduce-reduce conflicts

- Can CFG be used for semantic analysis?

  No. Semantic analysis relies on context, which cannot be described by CFG.

Compilation Phases[编译阶段]



src → AST: Example

ir_test.c

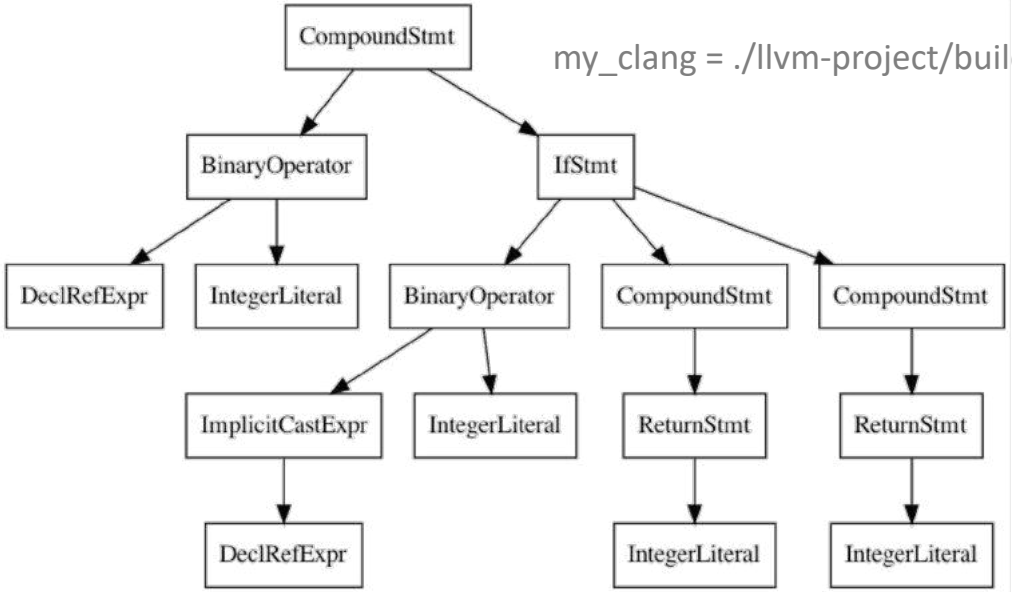
```
1 int a;
2
3 int main() {
4   a = 3;
5
6   if (a > 0) {
7     return 1;
8   } else {
9     return 0;
10  }
11 }
```

$<*my_clang*> -cc1 -ast-view ir_test.c

$dot -Tpng -o ir_test.png ir_test.dot

my_clang = ./llvm-project/build/bin/clang

AST



AST → IR: Example

$<*my_clang*> -Xclang -ast-dump -fsyntax-only ir_test.c

```
TranslationUnitDecl 0x1d2654a8 <<invalid sloc>> <invalid sloc>
... cutting out internal declarations of clang ...
|-VarDecl 0x13800f670 <ir_test.c:1:1, col:5> col:5 used a 'int'
`-FunctionDecl 0x13800f778 <line:3:1, line:11:1> line:3:5 main 'int ()'
  `-CompoundStmt 0x13800f9a8 <col:12, line:11:1>
```

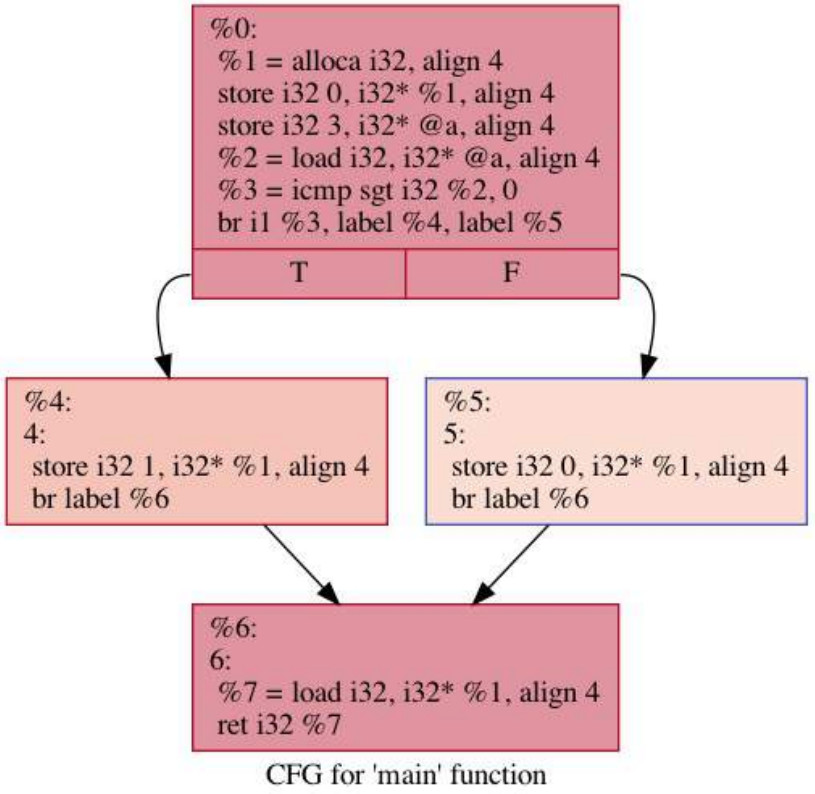
$<*my_clang*> -emit-llvm -S ir_test.c

```
; ModuleID = 'ir_test.c'
source_filename = "ir_test.c"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-macosx11.0.0"

@a = global i32 0, align 4

; Function Attrs: noinline nounwind optnone ssp uwtable(sync)
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  store i32 0, ptr %retval, align 4
  store i32 3, ptr @a, align 4
  %0 = load i32, ptr @a, align 4
  %cmp = icmp sgt i32 %0, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  store i32 1, ptr %retval, align 4
  br label %return

if.else:                                          ; preds = %entry
  store i32 0, ptr %retval, align 4
  br label %return

return:                                           ; preds = %if.else, %if.then
  %1 = load i32, ptr %retval, align 4
  ret i32 %1
}

attributes #0 = { noinline nounwind optnone ssp uwtable(sync) "frame-pointer"="non-leaf" "no-trapping-math"="true" "s
otprod,+fp-armv8,+fp16fml,+fullfp16,+lse,+neon,+ras,+rcpc,+rdm,+sha2,+sha3,+v8.1a,+v8.2a,+v8.3a,+v8.4a,+v8.5a,+v8a,+z

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"uwtable", i32 1}
!3 = !{i32 7, !"frame-pointer", i32 1}
!4 = !{!"clang version 17.0.0 (https://github.com/llvm/llvm-project.git f759275c1c8ed91f19a6b8db228115c7f75d460b)"}
```

AST → IR: Example (cont.)

$clang -emit-llvm -S ir_test.c

```
; ModuleID = 'ir_test.c'
source_filename = "ir_test.c"
target datalayout = "e-m:e-i8:8:32-i16:16:32-i64:64-i128:128-n32:64-S128
target triple = "aarch64-unknown-linux-gnu"

@a = dso_local global i32 0, align 4

; Function Attrs: noinline nounwind optnone
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 3, i32* @a, align 4
  %2 = load i32, i32* @a, align 4
  %3 = icmp sgt i32 %2, 0
  br i1 %3, label %4, label %5

4:                                                ; preds = %0
  store i32 1, i32* %1, align 4
  br label %6

5:                                                ; preds = %0
  store i32 0, i32* %1, align 4
  br label %6

6:                                                ; preds = %5, %4
  %7 = load i32, i32* %1, align 4
  ret i32 %7
}

attributes #0 = { noinline nounwind optnone "correctly-rounded-divide-sqrt
" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="fa
features"="+neon" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"Debian clang version 11.0.1-2"}
```



CFG for 'main' function

$dot -Tpng -o ir_test.png .ir_test.dot

$opt -dot-cfg ir_test.ll

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.)

```
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.
```

```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;
  }
};
```

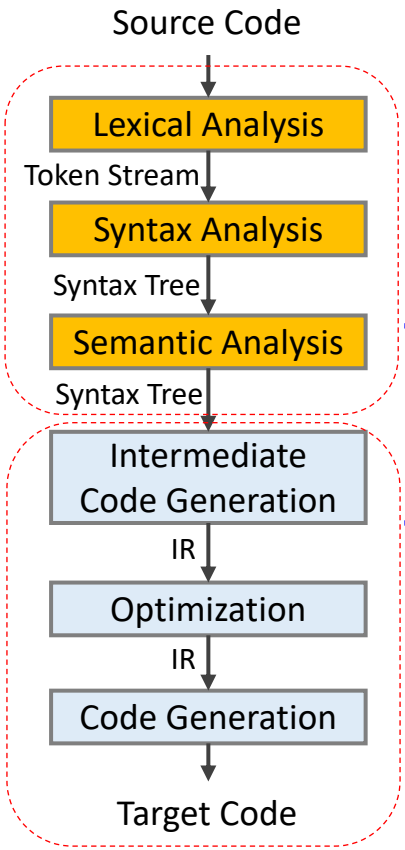

```
test.cpp:6:27: error: expected class-name before '{' token
    6 | class Child : public Base {
      |                           ^
test.cpp:15:8: error: 'void Child::doSomething()' cannot be overloaded with 'void Child::doSomething()'
   15 |   void doSomething() {
      |        ^~~~~~~~~~~
test.cpp:9:8: 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

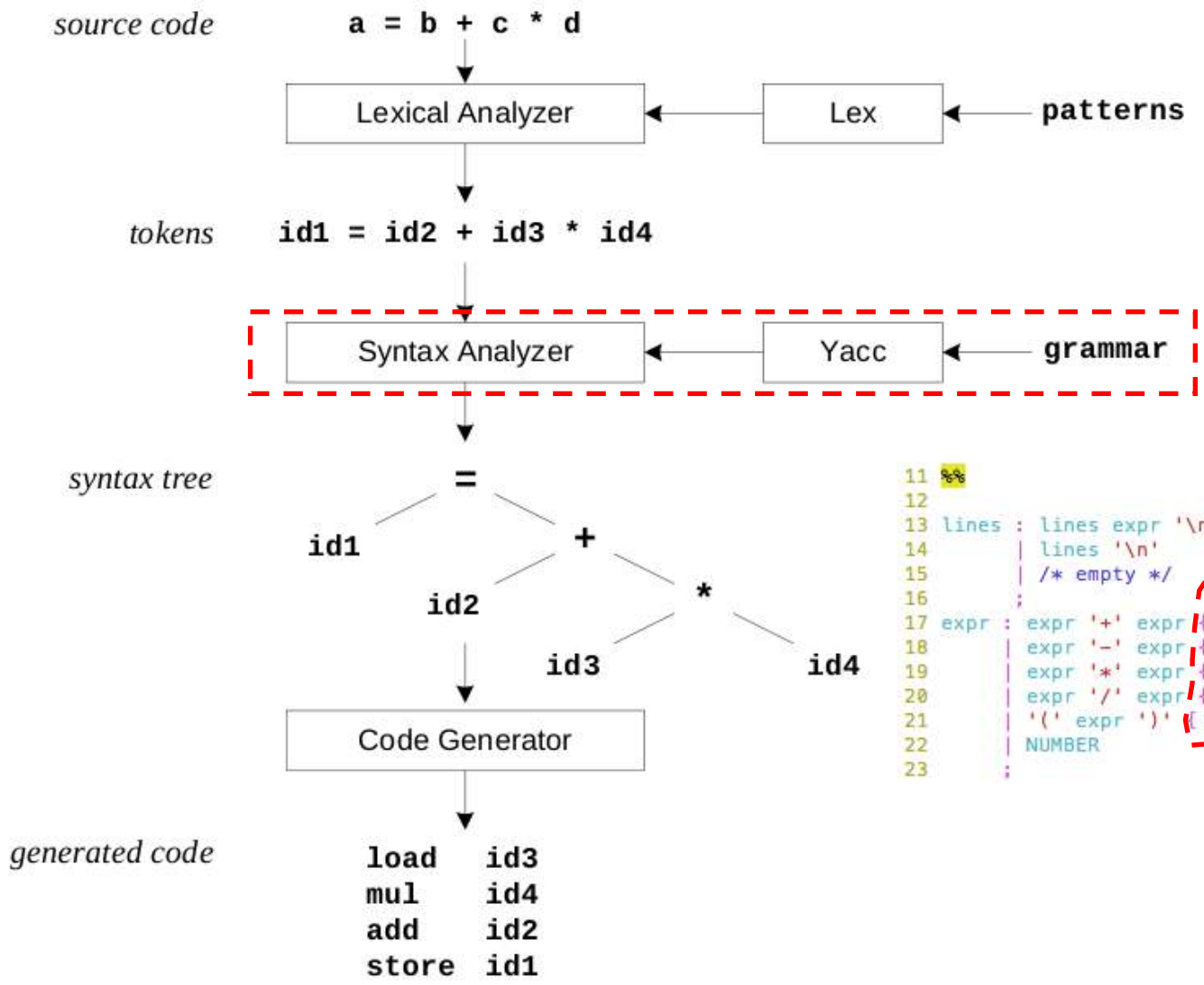
Syntax Directed Translation[语法制导翻译]



Semantic Translation
(语义翻译)

Syntax Directed Translation
(语法制导翻译)

Semantic in Practice

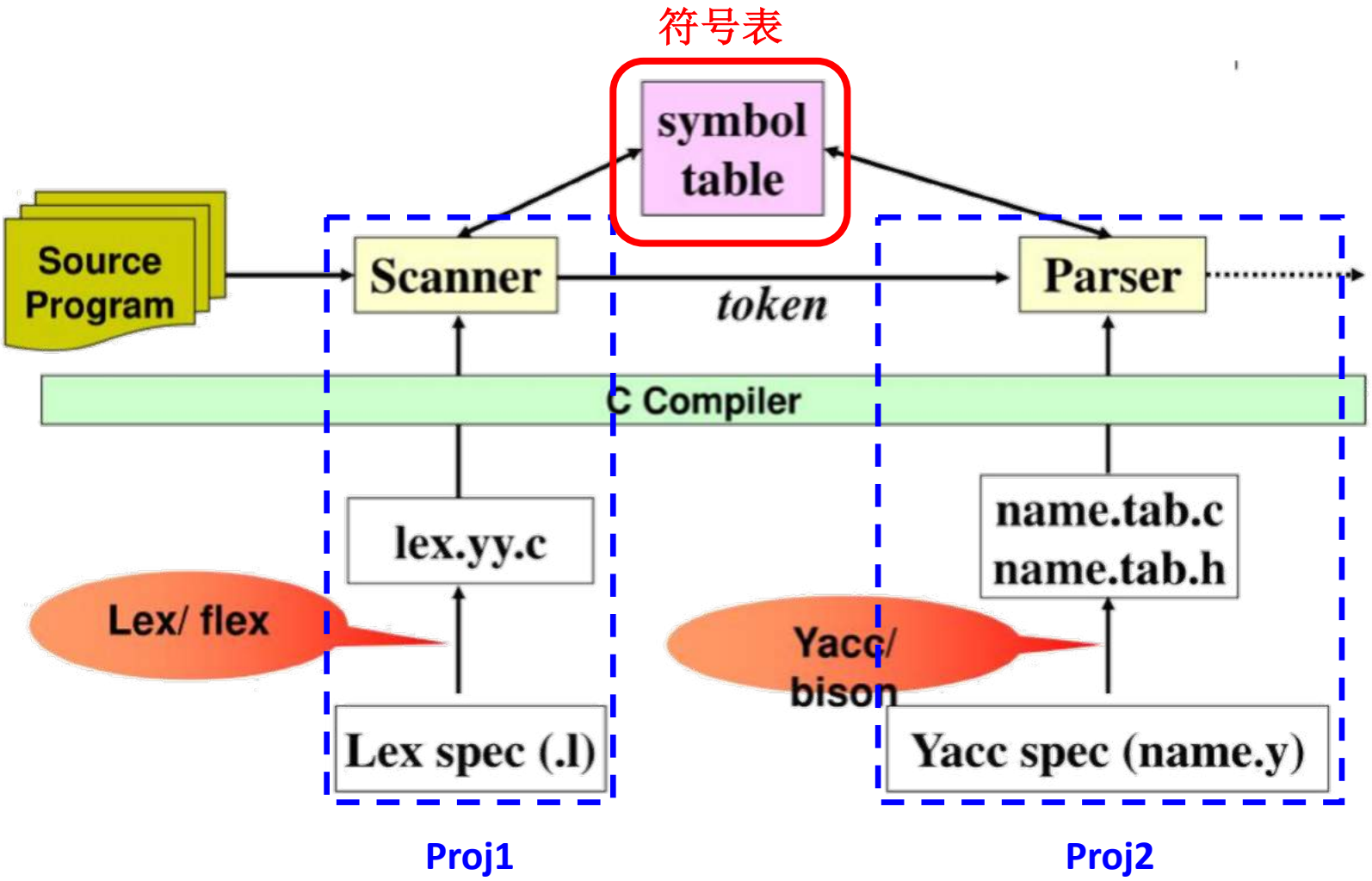


Semantic Actions

```
11 %%
12
13 lines : lines expr '\n' { printf("= %g\n", $2); }
14       | lines '\n'
15       | /* empty */
16       ;
17 expr : expr '+' expr { $$ = $1 + $3; }
18      | expr '-' expr { $$ = $1 - $3; }
19      | expr '*' expr { $$ = $1 * $3; }
20      | expr '/' expr { $$ = $1 / $3; }
21      | '(' expr ')' { $$ = $2; }
22      | NUMBER
23      ;
```


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[语义分析阶段填充]
- Symbol table is used in code generation to output assembler directives of the appropriate size & type[后续代码生成阶段使用]

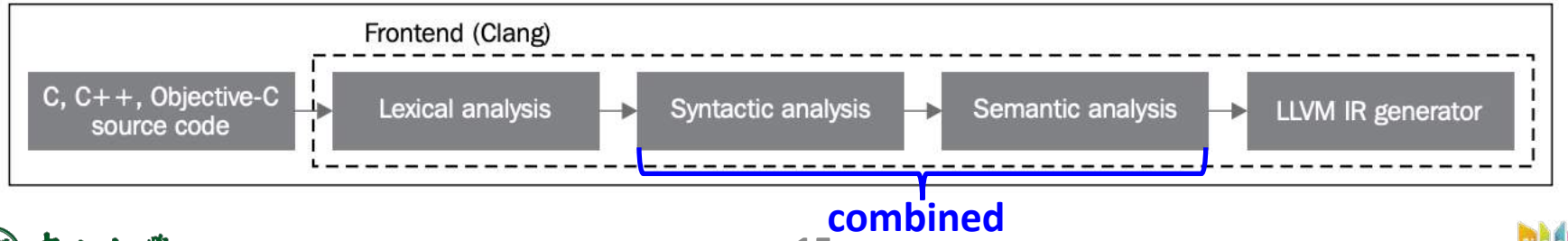
LLVM: Semantic Analysis

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

```
1202  StmtResult Parser::ParseIfStatement(SourceLocation *TrailingElseLoc) {

1341    // perform semantic checking for the if statement, emitting diagnostics accordingly
1342    return Actions.ActOnIfStmt(IfLoc, IsConstexpr, InitStmt.get(), Cond,
1343                               ThenStmt.get(), ElseLoc, ElseStmt.get());
1344  }
```

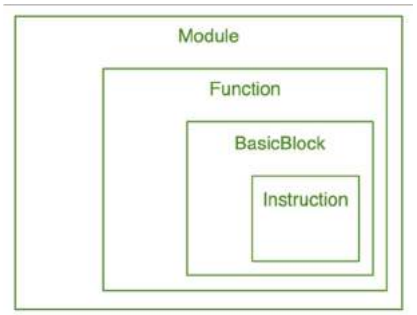
https://github.com/llvm-mirror/clang/blob/master/lib/Parse/ParseStmt.cpp
https://clang.llvm.org/doxygen/ParseAST_8cpp_source.html

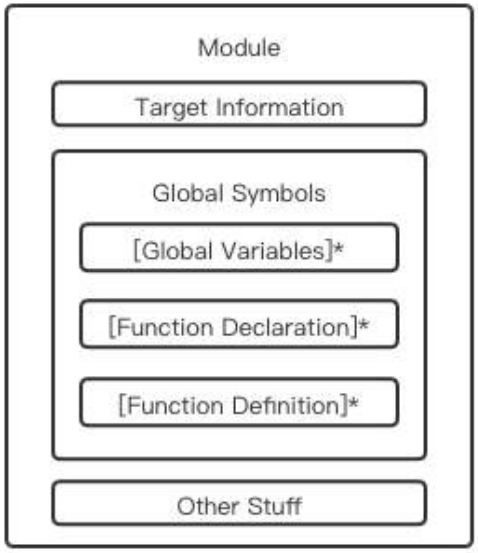
  - After the combined parsing and semantic analysis, the *ParseAST* function invokes the method *HandleTranslationUnit* to trigger any client that is interested in consuming the final AST

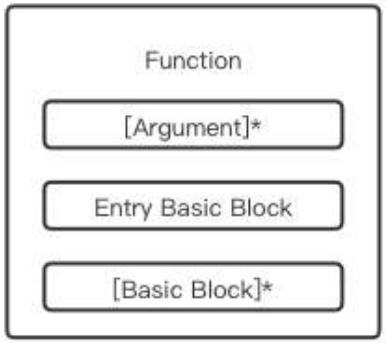


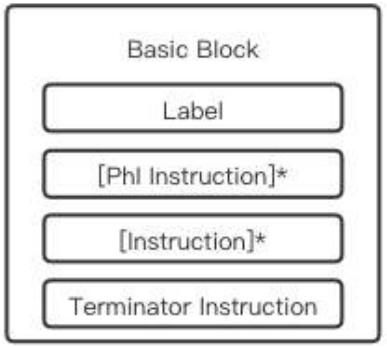
LLVM: Module

- 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

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
     ;
```

```
declarator
  : IDENTIFIER
    {
      $$ = par::gMgr.make<asg::VarDecl>();
      $$->name = std::move(*$1);
      delete $1;

      // 插入符号表
      par::Symtbl::g->insert_or_assign($$->name, $$);
    }
```

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

    E -> 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 -> E_1 + E_2 {E.val = E_1.val + E_2.val}

    E -> id {E.val = id.lexval}

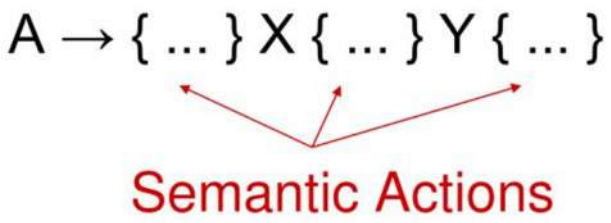
SDD vs. SDT

- 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 progs 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

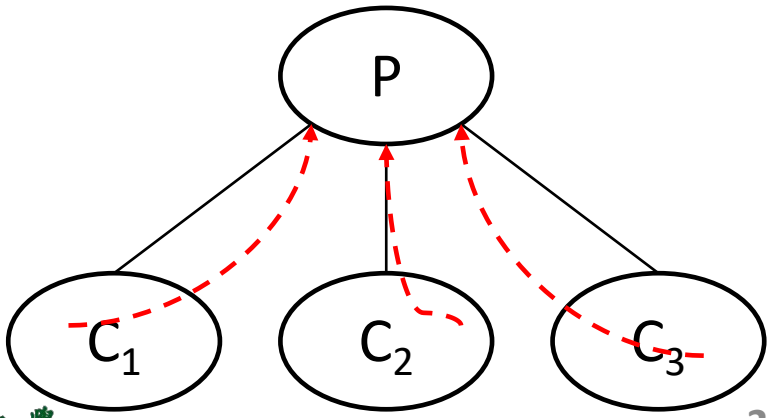


SDD[语法制导定义]

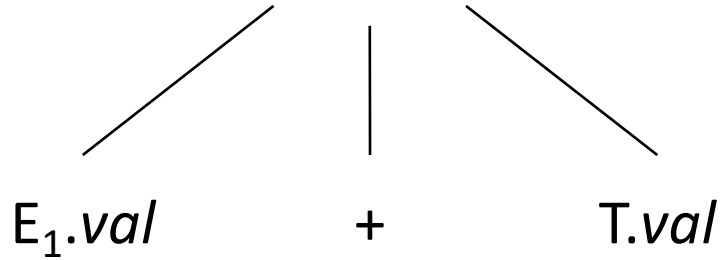
- SDD has two types of attributes[两种属性]
  - For a non-terminal *A* at a parse-tree node *N*
- **Synthesized attribute**[综合属性]
  - Defined by a semantic rule associated with the production at *N*
    - The production must have *A* as its head (i.e., A -> …)
  - A synthesized attribute of node *N* is defined 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]

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

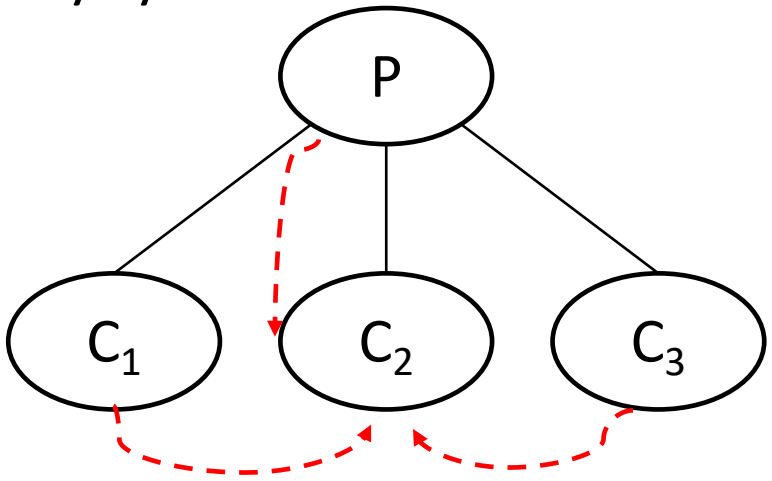


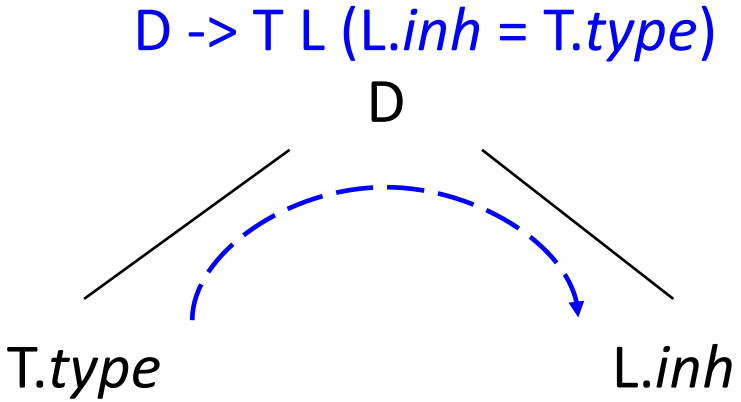
E.*val* = E_1.*val* + T.*val*



Inherited Attribute[继承属性]

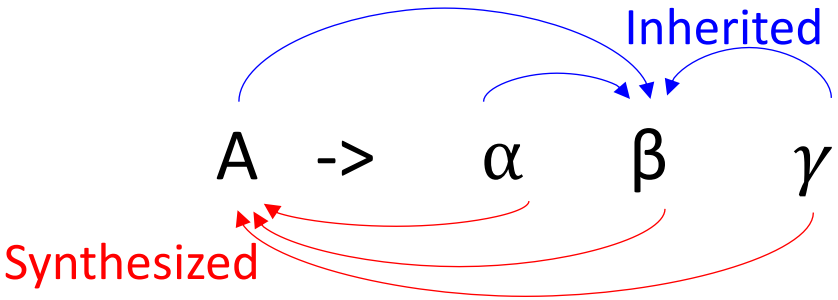
- Inherited attribute for non-terminal *A* of parse-tree node *N*[非终结符继承属性]
  - Only defined by *N*'s parent, *N*'s siblings and *N* itself
    - Passed down a parse tree
  - C_2.inh_attr = f(P.attrs, C_1.attrs, C_2.attrs, C_3.attrs)
- Terminals cannot have inherited attributes[终结符无继承属性]
  - Only synthesized attributes from lexical analysis





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.)

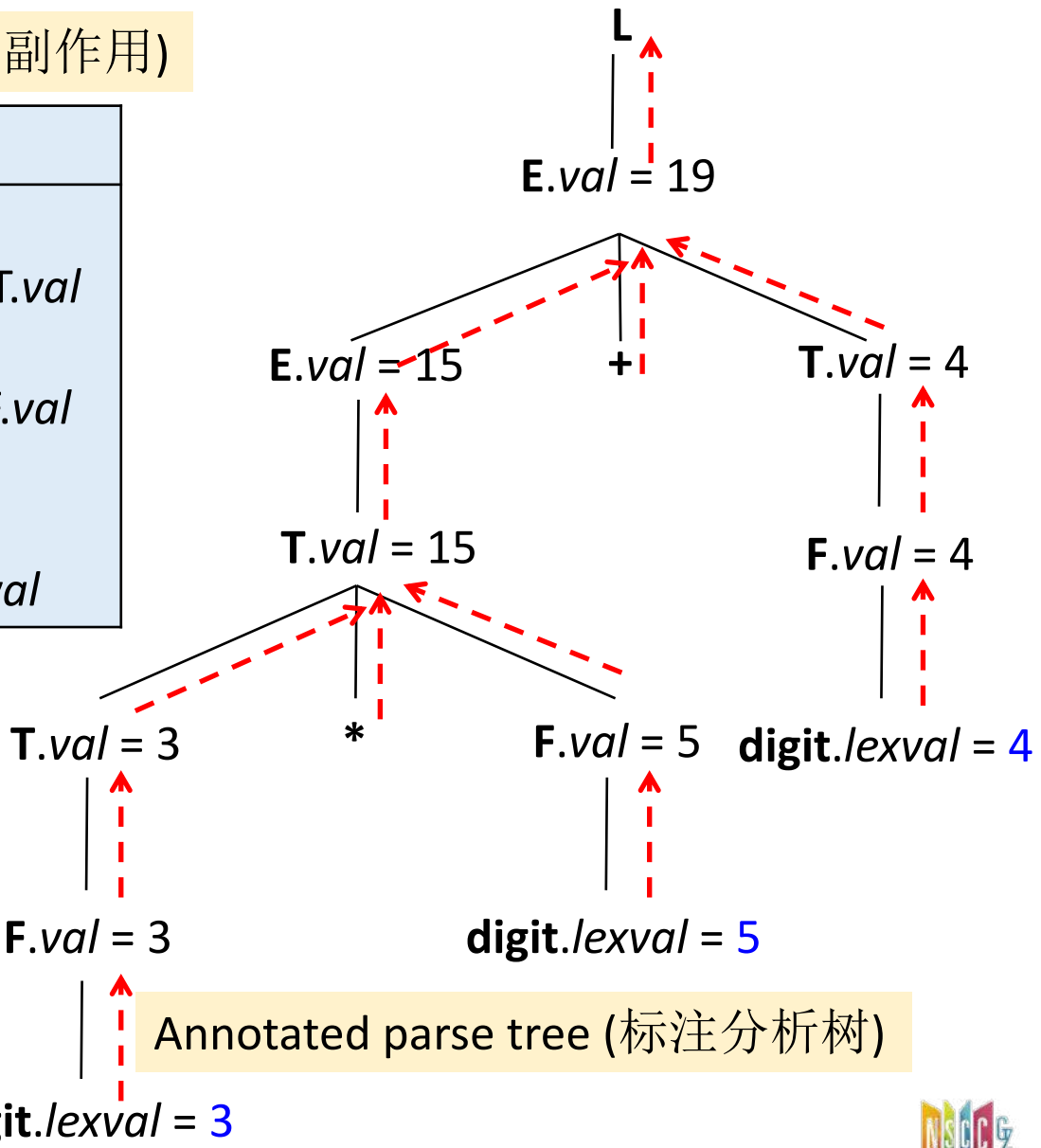
SDD:

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)

Example: Inherited Attribute[继承]

SDD:

Production Rules	Semantic Rules
(1) D -> T L	L.*inh* = T.*type*
(2) T -> int	T.*type* = int
(3) T -> float	T.*type* = float
(4) L -> L_1, id	L_1.*inh* = L.*inh*
	addtype(id.*entry*, L.*inh*)
(5) L -> id	addtype(id.*entry*, L.*inh*)

T has synthesized attribute *type*

L has inherited attribute *inh*

Pointing to a symbol-table[符号表] object

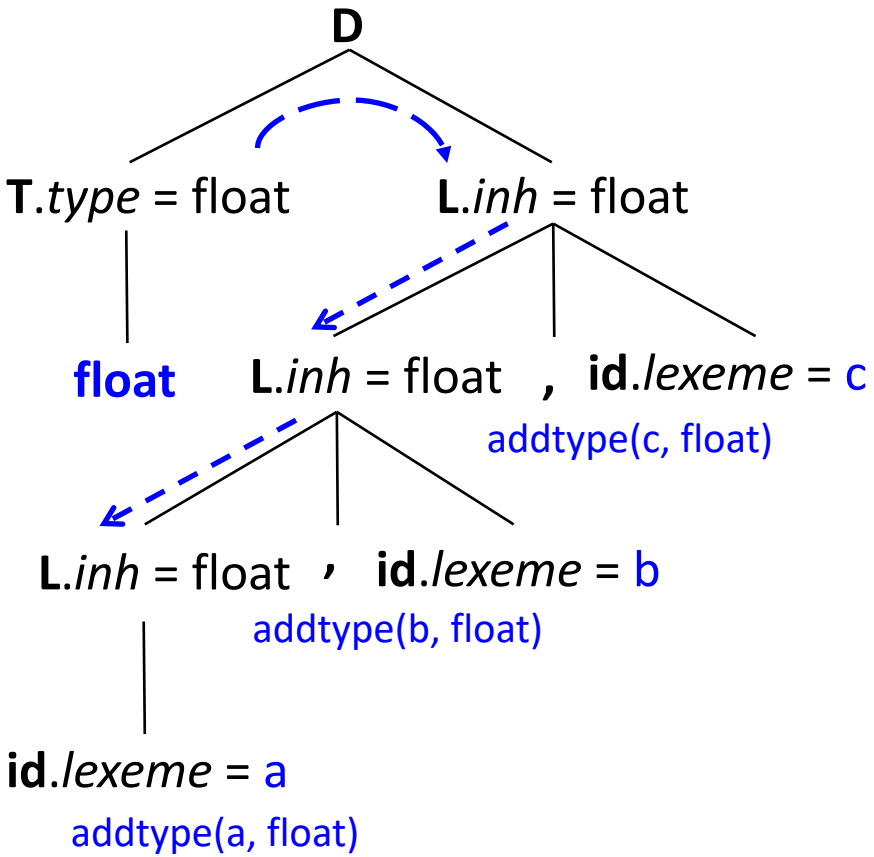
Variable declaration of type int/float followed by a list of IDs:

(1) Declaration: a type *T* followed by a list of *L* identifiers
(2) Evaluate the synthesized attribute *T.type* (int)
(3) Evaluate the synthesized attribute *T.type* (float)
(4) Pass down type, and add type to symbol table entry for the identifier
(5) Add type to symbol table

Example: Inherited Attribute (cont.)

SDD:

Production Rules	Semantic Rules
(1) D -> T L	L.*inh* = T.*type*
(2) T -> int	T.*type* = int
(3) T -> float	T.*type* = float
(4) L -> L_1, id	L_1.*inh* = L.*inh* addtype(id.*entry*, L.*inh*)
(5) L -> id	addtype(id.*entry*, L.*inh*)



Input:

float a, b, c

type depends on child

inh depends on sibling or parent

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]
  - Can also have actions being annotated[也可标注语义动作]

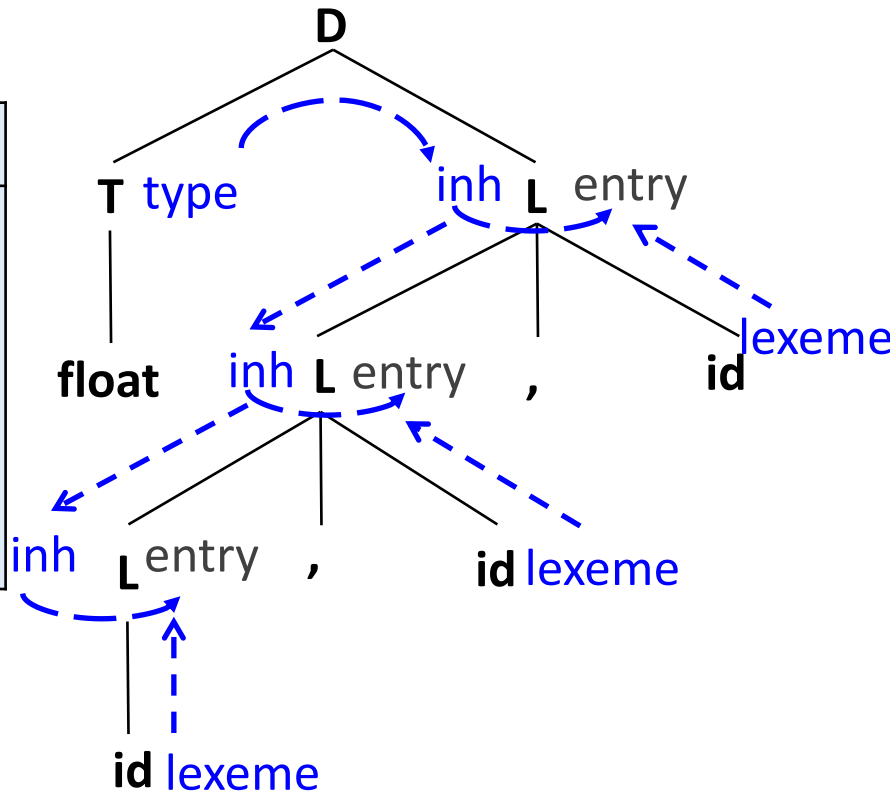
Dependence Graph[依赖图]

- 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[按照依赖顺序计算]
- **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[依赖图决定属性值的计算]
  - Depicts the flow of info among the attribute instances in a particular parse tree[描绘了分析树的属性信息流]
    - **Directed graph** where edges are dependence relationships between attributes
    - For each parse-tree node *X*, there's a graph node for each attr of *X*
    - If attr *X.a* depends on attr *Y.b*, then there's one directed edge from *Y.b* to *X.a*

Example: Dependency Graph

SDD:

Production Rules	Semantic Rules
(1) D -> T L	L.*inh* = T.*type*
(2) T -> int	T.*type* = int
(3) T -> float	T.*type* = float
(4) L -> L_1, id	L_1.*inh* = L.*inh*
	addtype(id.*entry*, L.*inh*)
(5) L -> id	addtype(id.*entry*, L.*inh*)



Input:

float a, b, c

'entry' is dummy attribute for the *addtype()*

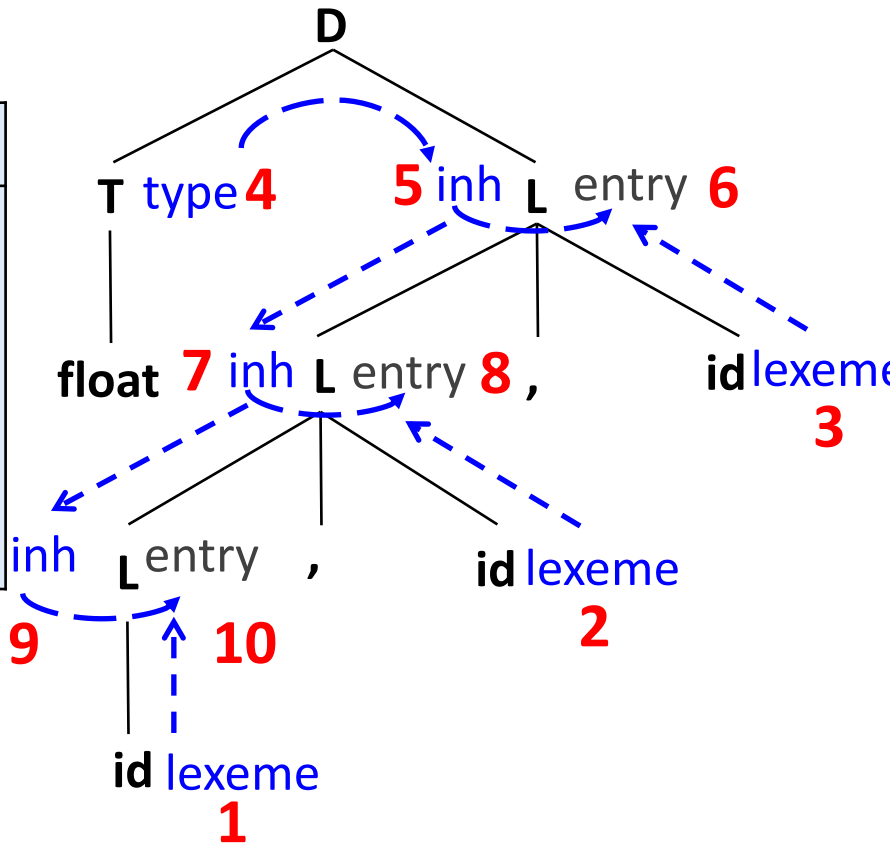
Evaluation Order[属性值计算顺序]

- Ordering the evaluation of attributes[计算顺序]
  - Dependency graph characterizes possible orders in which we can evaluate the attributes at the various nodes of a parse-tree
- 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, …, N_k such that if there is an edge of the graph from N_i to N_j, then i < j
  - Such an ordering embeds a directed graph into a linear order, and is 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:

Production Rules	Semantic Rules
(1) D -> T L	L.*inh* = T.*type*
(2) T -> int	T.*type* = int
(3) T -> float	T.*type* = float
(4) L -> L_1, id	L_1.*inh* = L.*inh*
	addtype(id.*lexeme*, L.*inh*)
(5) L -> id	addtype(id.*lexeme*, L.*inh*)



Input:

float a, b, c

Topological sort:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10