Compilation Principle
编 译 原 理

第11讲：语法分析(8)

张献伟

xianweiz.github.io

DCS290, 4/6/2021

Review Questions (1)

- Why LR(0) is of limited usage?

  No lookahead, easy to have shift-reduce and reduce-reduce conflicts

- How does SLR(1) improve LR(0)?

  Lookahead using the Follow set when reduce happens

- Why we further use LR(1)?

  Follow set is not precise enough, still easy to have conflicts

- At high level, how does LR(1) improve SLR(1)?

  Splitting Follow set (i.e., splitting states) to enforce reduce to consider not only the stack top

- How does LR(1) split the states?

  Add lookaheads to each item, i.e., LR(1) item=LR(0) item+lookahead

Review Questions (2)

- How to understand the item [A -> u•, a/b/c]

  Reduce only using A -> u, when the next input symbol is a/b/c

- Then, what are the drawbacks of LR(1)?

  More states because of the splitting, further much larger parse table

- What is LALR(1)?

  LookAhead LR. A compromise between LR(1) and LR(0)/SLR(1)

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

  Merge similar states to reduce table rows

- LR(0) -> SLR(1) -> LR(1), what is trend of improvement?

  Reduce action is more and more precise

State Merging[状态合并]

- Merge states with the same core[同心]
  - Core: LR(1) items minus the lookahead (i.e., LR(0) items)
  - All items are identical except lookahead

I_3: X → a·X, a/b X → .aX, a/b X → ·b, a/b	I_6: X → a·X, \$ X → .aX, \$ X → ·b, \$	→	I_{36}: X → a·X, a/b/\$ X → .aX, a/b/\$ X → ·b, a/b/\$
I_4: X → b·, a/b	I_7: X → b·, \$	→	I_{47}: X → b·, a/b/\$
I_8: X → aX·, a/b	I_9: X → aX·, \$	→	I_{89}: X → aX·, a/b/\$

State Merging (cont.)

State	ACTION			GOTO	
	a	b	$	S	X
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

LR(1)

State	ACTION			GOTO	
	a	b	$	S	X
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

LALR(1)

Merge Effects

- Merging of states can introduce **conflicts**[引入冲突]
  - Cannot introduce shift-reduce (s-r) conflicts
    - i.e., a s-r conflict cannot exist in a merged set unless the conflict was already present in one of the original LR(1) sets
  - Can introduce reduce-reduce (r-r) conflicts
    - LR was introduced to split the Follow set on reduce action
    - Merging reverts the splitting

- **Detection of errors** may be delayed[推迟错误识别]
  - On error, LALR parsers will not perform shifts beyond an LR parser, but may perform more reductions before finding error
  - We'll see an example

Merge Conflict: Shift-Reduce

- Shift-reduce conflicts are **not** introduced by merging

- Suppose

  Sij: [A -> $\alpha\cdot$, a] reduce on input a

  [B -> β.aσ, b] shift on input a

  Formed by merging Si and Sj

- Cores must be the same for Si and Sj, and thus one of them must contain [A -> $\alpha\cdot$, a] and [B -> β.aσ, b]
  - Shift-reduce conflicts were already present in either Si and Sj (or both) and not newly introduced by merging

SUN YAT-SEN UNIVERSITY

Merge Conflict: Reduce-Reduce

- Reduce-reduce conflicts can be introduced by merging

S' –> S
S –> aBc | bCc | aCd | bBd
B –> e
C –> e

I_{69}:
C → e·, c/d
B → e·, d/c

Reduce to B or C when next token is c or d

I_0: S' -> •S, \$
S -> •aBc, \$
S -> •bCc, \$
S -> •aCd, \$
S -> •bBd, \$

I_1: S' -> S•, \$

I_2: S -> a•Bc, \$
S -> a•Cd, \$
B -> •e, c
C -> •e, d

I_3: S -> b•Cc, \$
S -> b•Bd, \$
C -> •e, c
B -> •e, d

I_4: S -> aB•c, \$

I_5: S -> aC•d, \$

I_6: B -> e•, c
C -> e•, d

I_7: S -> bC•c, \$

I_8: S -> bB•d, \$

I_9: B -> e•, d
C -> e•, c

I_{10}: S -> aBc•, \$

I_{11}: S -> aCd•, \$

I_{12}: S -> bCc•, \$

I_{13}: S -> bBd•, \$

Example: Error Delay

(0) S' -> S
(1) S -> XX
(2) X -> aX
(3) X -> b

Input: aab$

State	ACTION			GOTO	
	a	b	$	S	X
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

state → S_0
symbol → $ | aab$

state → S_0S_3
symbol → $ a | ab$

state → $S_0S_3S_3$
symbol → $ a a | b$

state → $S_0S_3S_3S_4$
symbol → $ a a b | $

Example: Error Delay (cont.)

(0) S' -> S
(1) S -> XX
(2) X -> aX
(3) X -> b

Input: aab$

State	ACTION			GOTO	
	a	b	$	S	X
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

state → S_0
symbol → $ | aab$

state → S_0S_{36}
symbol → $ a | ab$

state → $S_0S_{36}S_{36}$
symbol → $ a a | b$

state → $S_0S_{36}S_{36}S_{47}$
symbol → $ a a b | $

state → $S_0S_{36}S_{36}S_{89}$
symbol → $ a a X | $

state → $S_0S_{36}S_{89}$
symbol → $ a X | $

state → S_0S_2
symbol → $ X | $

LALR Table Construction[解析表构建]

- LALR(1) parsing table is built from the configuration sets in the same way as LR(1)[同样方法构建的项目集]
  - The lookaheads determine where to place reduce actions
  - If there are no mergable states, the LALR(1) table will be identical to the LR(1) table and we gain nothing
  - Usually, there will be states that can be merged and the LALR table will thus have **fewer rows** than LR
- LALR(1) table have the same #states (rows) with SLR(1) and LR(0), but have fewer reduce actions[同等数目的状态，但更少的规约动作]
  - Some reductions are not valid if we are more precise about the lookahead
  - Some conflicts in SLR(1) and LR(0) are avoided by LALR(1)

LALR Table Construction (cont.)

- Brute force[暴力方式]
  - Construct LR(1) states, then merge states with same core
  - If no conflicts, you have a LALR parser
  - Inefficient: building LR(1) items are expensive in time and space
    - We need a better solution

- Efficient way[高效方式]
  - Avoid initial construction of LR(1) states
  - Merge states on-the-fly (step-by-step merging)
    - States are created as in LR(1)
    - On state creation, immediately merge if there is an opportunity

LALR(1) Grammars

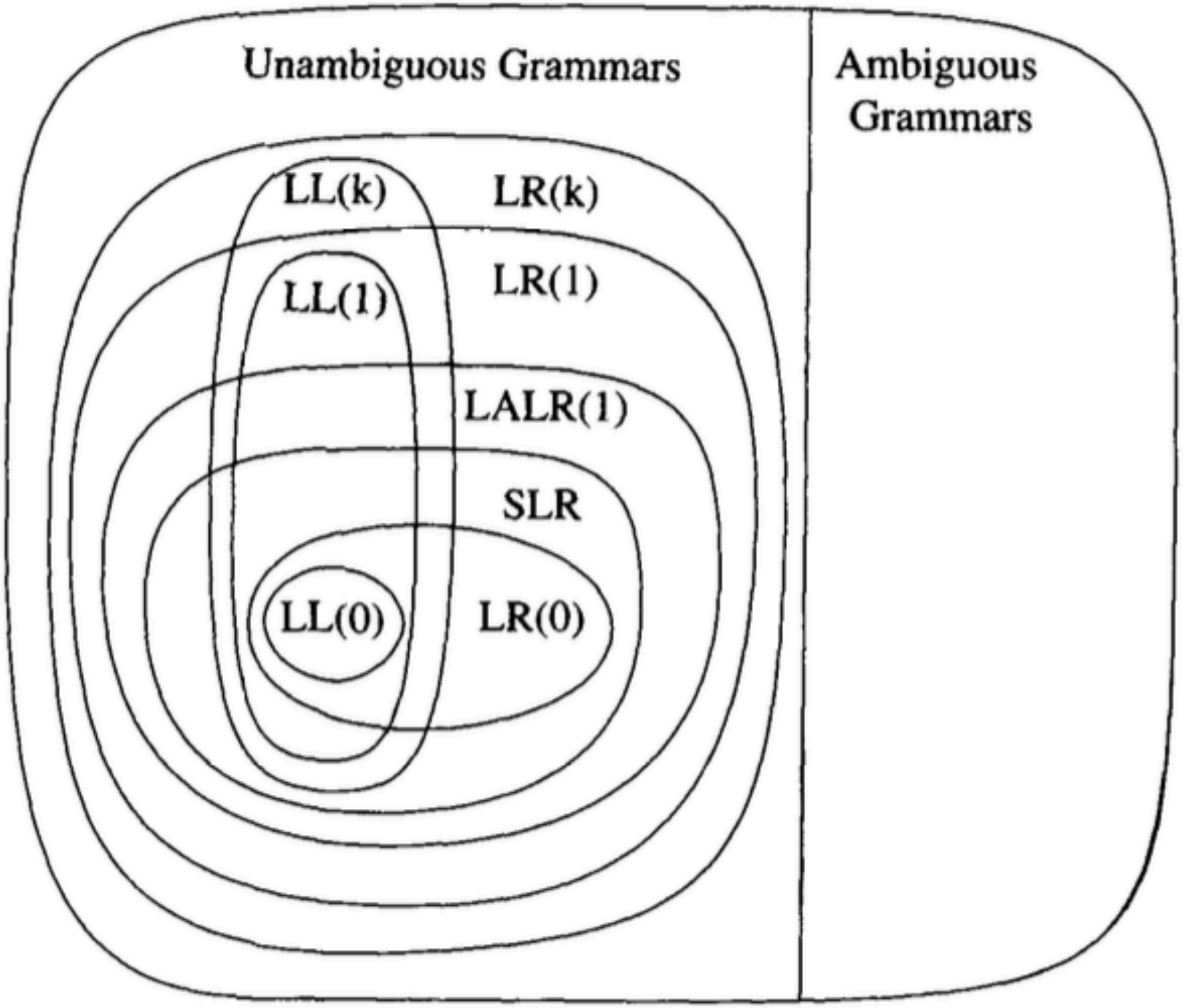
- For a grammar, if the LALR(1) parse table has no conflicts, then we say the grammar is LALR(1)
  - No formal definition of a set of rules
- LALR(1) is a <u>subset of LR(1)</u> and a <u>superset of SLR(1)</u>
  - A SLR(1) grammar is definitely LALR(1)
  - A LR(1) grammar may or may not be LALR(1)
    - Depends on whether merging introduces conflicts
  - A non-SLR(1) grammar may be LALR(1)
    - Depends on whether the more precise lookaheads resolve the SLR(1) conflicts
- LALR(1) reaches a good balance between the **lookahead power** and the **table size**
  - Most used variant of the LR family

LL vs. LR Parsing (LL < LR)

- LL(k) parser, each expansion A -> α is decided based on
  - Current non-terminal at the top of the stack
    - Which LHS to produce
  - k terminals of lookahead at *beginning* of RHS
    - Must guess which RHS by peeking at first few terminals of RHS

- LR(k) parser, each production A -> α· is decided based on
  - RHS at the top of the stack
    - Can postpone choice of RHS until entire RHS is seen
    - Common left factor is OK – waits until entire RHS is seen anyway
    - Left recursion is OK – does not impede forming RHS for reduction
  - k terminals of lookahead *beyond* RHS
    - Can decide on RHS after looking at entire RHS plus lookahead

Hierarchy of Grammars[文法层级]



总结: 语法分析（1）

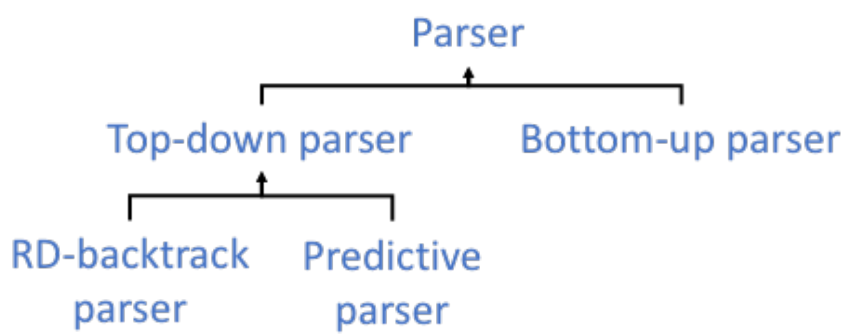
- **语法分析**(Syntax analysis)是编译的第二个阶段
  - 输入: 词法分析产生的token序列
  - 输出: 分析树(parse tree)或抽象语法树(abstract syntax tree ,AST)
- **语法指定**(Syntax specification)
  - 词法分析使用的RE/FA表达能力不够(e.g., 嵌套结构)
  - 需要使用文法(grammar), 尤其是上下文无关文法(context-free grammar, CFG)
- **文法形式化定义**: {T, N, s, σ}
  - T: terminal symbols[终结符] = 词法分析的token, 分析树的叶子节点
  - N: non-terminal symbols[非终结符], 分析树的内部节点
  - s: start symbol[开始符号]
  - σ: set of productions[产生式], 形式：LHS -> RHS

总结: 语法分析（2）

- 推导(Derivation)
  - 对产生式的若干次使用 (从LHS到RHS)
    - 从文法开始符号到输入串(input string)
- 归约(Reduce)
  - 推导的逆过程(从RHS到LHS)
    - 从输入串(input string)到开始符号
- 分析树(Parse tree)
  - 是推导的图形化表示，略去了推导中产生式的使用顺序
- 歧义文法(Ambiguous grammar)
  - 某个句子对应多个(最左或最右)分析树
  - 通过指定优先级(precedence)和和结合性(associativity)来改写文法以消除歧义

总结: 语法分析（3）

- 语法分析(或解析)就是处理给定文法的输入句子，构建一个以分析树或抽象语法树表示的推导
  - **自顶向下(Top-down)**: 从根节点扩展到叶子节点，每步考虑
    - 替换哪个非终结符?
    - 使用哪个产生式来替换?
  - **自底向上(Bottom-up)**: 从叶子节点回到根节点
    - 消耗输入token还是归约?
    - 使用哪个产生式来归约?



总结: 语法分析（4）

- Top-down分析
  - 递归下降分析(Recursive descent): 试错->回溯(backtracking)
    - 消除左递归(Left recursion)
  - 预测分析(Predictive): 预测，无需回溯
    - 消除左递归，提取左共因子(Left factoring)

- 表驱动的LL(1)分析器
  - 四部分：input buffer, stack, parse table, parser driver
  - 基于<stack top, current token>来采取操作(expand or match)
  - 解析表行为文法的非终结符、列为文法的终结符号及$
    - 单元格存放一个产生式或空
    - 表格是借助First和Follow集来构建的

总结: 语法分析（5）

- Bottom-up分析
  - 主要有移进(Shift)和归约(Reduce)两个动作
  - 实现上主要是LR类型分析器
    - 表格驱动，高效

- 表驱动的LR分析器
  - 四部分： input buffer, stack, parse table, parser driver
  - 基于栈顶来采取操作(shift or reduce)
    - 栈保存状态序列和每个状态关联的文法符号
  - 解析表包含Action和Goto两个子表
    - 表格是通过识别文法的可能项目集及转换(i.e., DFA)
    - LR(0) -> SLR(1) -> LR(1) -> LALR(1)

Compilation Principle
编 译 原 理

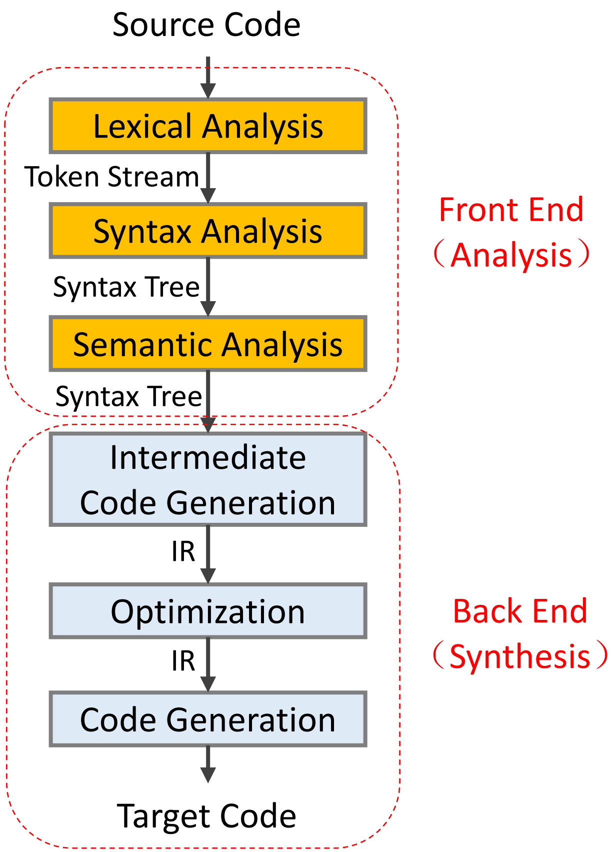
第11讲：语义分析(1)

张献伟

xianweiz.github.io

DCS290, 4/6/2021

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?
- 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?

Example

```
#include <iostream>

using namespace std;

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

  void doSomething() {
    int x[] = new string;
    x[5] = myInteger * y;
  }

  void doSomething() {
  }

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

base class not defined

wrong type

1) y variable not declared

2) cannot multiple a string

cannot redefine functions

cannot add void to int

no main() function

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."
- 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
  - Semantics: defines what the programs means (i.e., what each program does when it executes)

- 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:

    $\{wcw \mid w \in (a|b)^*\}$

    - 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
  - Compiler's last chance to reject incorrect programs
  - 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]
    - … …
- 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 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]