Compilation Principle
编 译 原 理

第12讲：语法分析(9)

张献伟

xianweiz.github.io

DCS290, 3/31/2022

Review Questions

- 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

- 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?

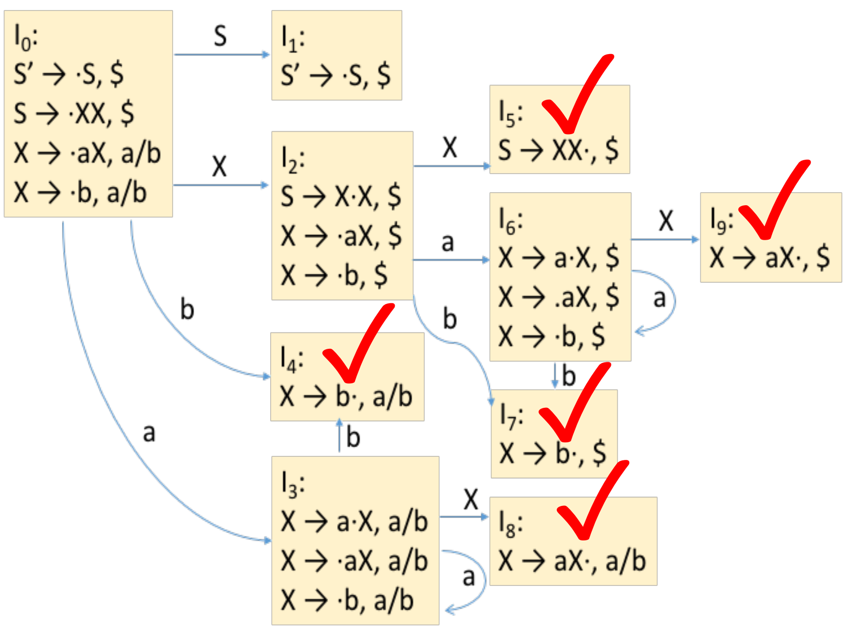
  Add lookaheads to each item, i.e., LR(1) item=LR(0) item+lookahead

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

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

Example

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



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		

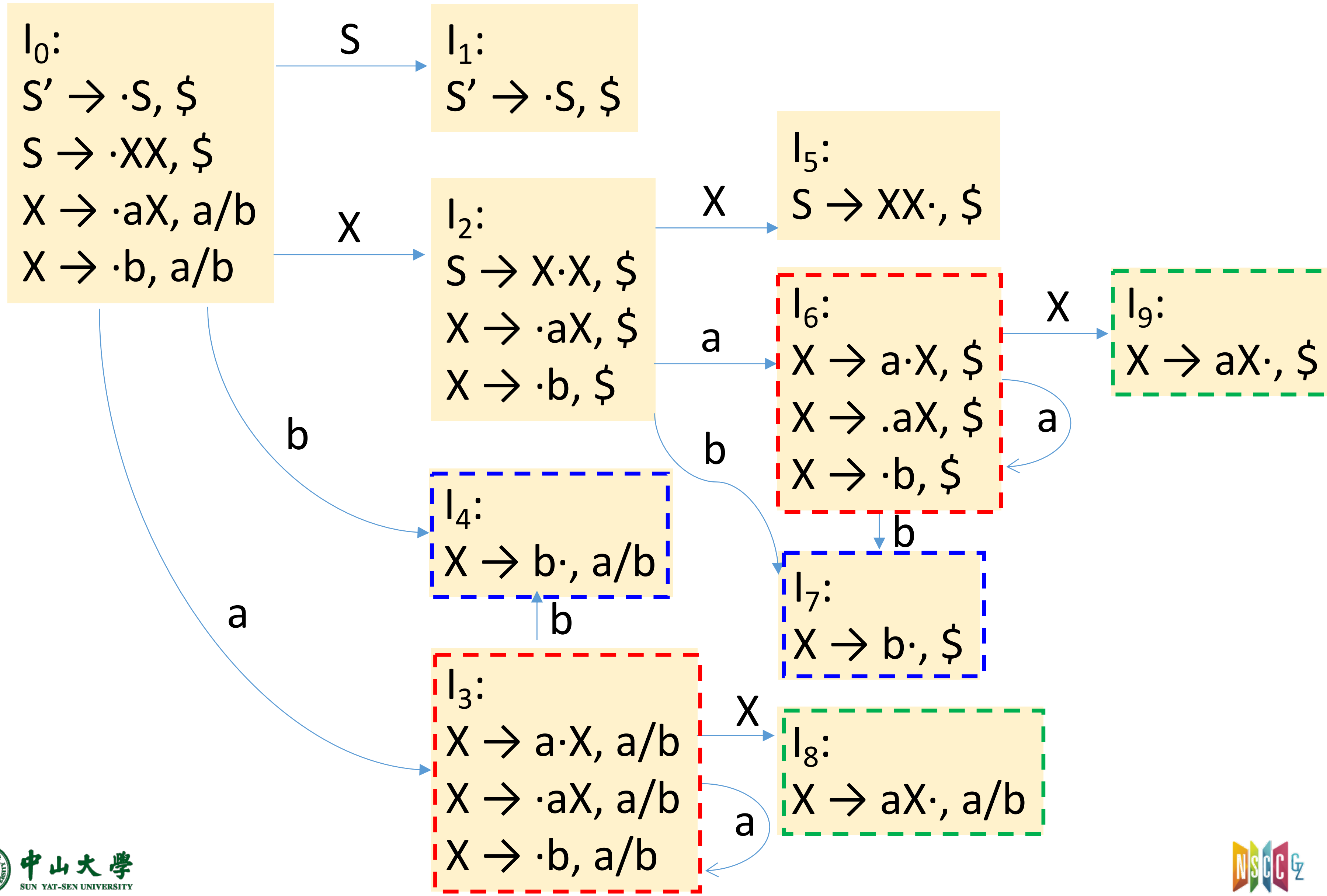
LR(1) Grammars

- Every SLR(1) grammar is LR(1), but the LR(1) parser may have more states than SLR(1) parser
  - LR(1) parser splits states based on differing lookaheads, thus it may avoid conflicts that would otherwise result if using the full Follow set
- A grammar is LR(1) if the following two conditions hold for each configurating set
  - (1) For any item [A → u·xv, a] in the set, with terminal *x*, there is no item in the set of form [B → v·, x]
    - In the table, this translates no shift-reduce conflict on any state
  - (2) The lookaheads for all complete items within the set must be disjoint, e.g. set cannot have both [A → u·, a] and [B → v·, a]
    - This translates to no reduce-reduce conflict on any state

LALR(1) Parser

- LR(1) drawbacks[缺点]
  - With state splitting, the LR(1) parser can have many more states than SLR(1) or LR(0) parser
    - One LR(0) item may split up to many LR(1) items
    - As many as all possible lookaheads
    - In theory can lead to an exponential increase in #states
- LALR (lookahead LR) – compromise LR(1) and SLR(1)[折衷]
  - Reduce the number of states in LR(1) parser by merging similar states[状态合并]
    - Reduces the #states to the same as SLR(1), but still retains the power of LR(1) lookaheads
  - Similar states: have same number of items, the core of each item is identical, and they differ only in their lookahead sets[移相似：核心相同，展望不同]

The Example



I0:
S' → ·S, $
S → ·XX, $
X → ·aX, a/b
X → ·b, a/b
S
I1:
S' → ·S, $
X
I2:
S → X·X, $
X → ·aX, $
X → ·b, $
X
I5:
S → XX·, $
a
I6:
X → a·X, $
X → .aX, $
X → ·b, $
X
I9:
X → aX·, $
a
b
b
I4:
X → b·, a/b
b
I7:
X → b·, $
b
a
I3:
X → a·X, a/b
X → ·aX, a/b
X → ·b, a/b
X
I8:
X → aX·, a/b
a

State Merging[状态合并]

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

I_3:	I_6:		I_{36}:
$X \to a \cdot X$, a/b	$X \to a \cdot X$, \$	⇒	$X \to a \cdot X$, a/b/\$
$X \to .aX$, a/b	$X \to .aX$, \$		$X \to .aX$, a/b/\$
$X \to \cdot b$, a/b	$X \to \cdot b$, \$		$X \to \cdot b$, a/b/\$

I_4:	I_7:		I_{47}:
$X \to b\cdot$, a/b	$X \to b\cdot$, \$	⇒	$X \to b\cdot$, a/b/\$

I_8:	I_9:		I_{89}:
$X \to aX\cdot$, a/b	$X \to aX\cdot$, \$	⇒	$X \to aX\cdot$, 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)

Grammar：

(0) S’ -> S

(1) S -> XX

(2) X -> aX

(3) X -> b

Merge Effects[合并效果]

- Merging of states can introduce **conflicts**[引入归约-归约冲突]
  - 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 contains: [A -> $\alpha\cdot$, a] reduce on input a

  [B -> $\beta.a\gamma$, b] shift on input a

  Formed by merging Si and Sj[注：Sij并不一定只有这两个item]
- Because
  - Cores must be the same for Si and Sj, and thus one of them must contain [A -> $\alpha\cdot$, a]
  - and it must have an item [B -> $\beta.a\gamma$, c] for some c
    - This state has the same shift/reduce conflict on a, i.e., the grammar was not LR(1)
  - Shift-reduce conflicts were already present in either Si and Sj (or both) and not newly introduced by merging

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[退化为LR(1)]
  - 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，但更少的归约动作]
  - 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)
  - For C language: SLR/LALR - 100s states, LR - 1000s states

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

LALR Summary[小结]

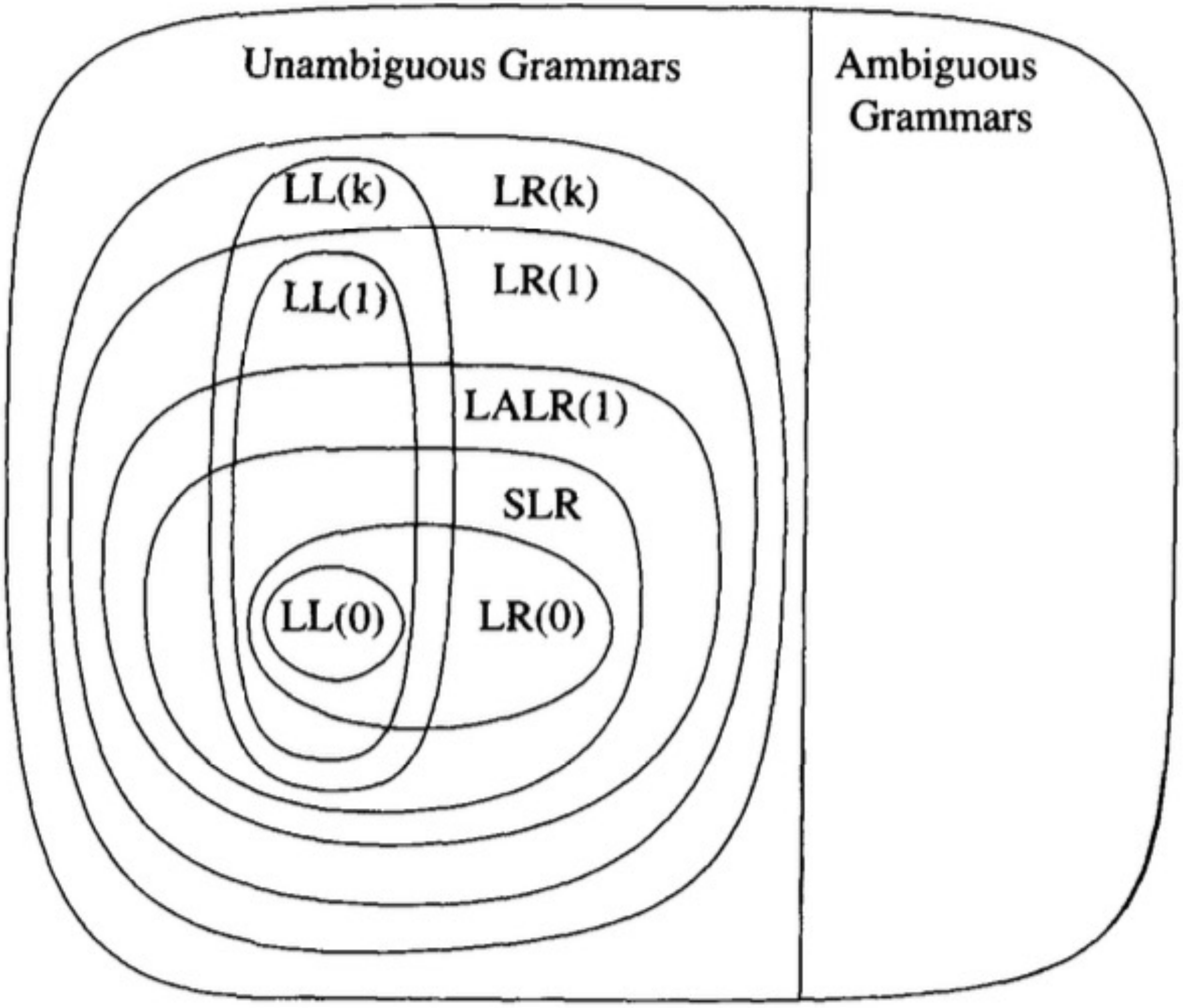
- LALR(1)是LR(1)和SLR(1)的平衡
  - 文法范围：LR > LALR > SLR
  - 状态数目：LR > LALR = SLR
- 假如一个文法G是LR而非SLR
  - 依靠Follow集进行归约不够精确 --> SLR产生了冲突
    - 而LR通过精确的lookahead解决了冲突
  - LALR对LR进行相似状态合并
    - 若合并后出现了冲突 --> 不是LALR文法
    - 若合并后没有冲突 --> 是LALR文法
      - LALR可以解析文法G，也即解决了SLR原有的冲突
      - 实际上LALR的状态数与SLR相同，但归约动作减少了（也即，对SLR解析表而言，多个移进/归约动作的单元格中的归约被消除了）
    - 如果没有相似状态，则LALR=LR
- 假如一个文法G是SLR
  - 那么G一定也是LR和LALR文法
  - LR的Follow集细分是不必要的，因此LALR合并回了SLR

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[依赖LHS]
    - Which LHS to produce
  - k terminals of lookahead at ***beginning*** of RHS[展望RHS]
    - Must guess which RHS by peeking at first few terminals of RHS

- LR(k) parser, each production A -> $\alpha\cdot$ is decided based on
  - RHS at the top of the stack[依赖RHS]
    - 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[超越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

中山大學 SUN YAT-SEN UNIVERSITY

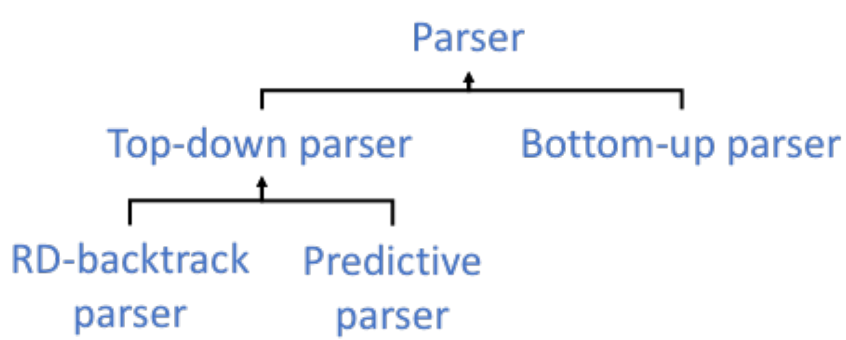
NSCC GZ

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

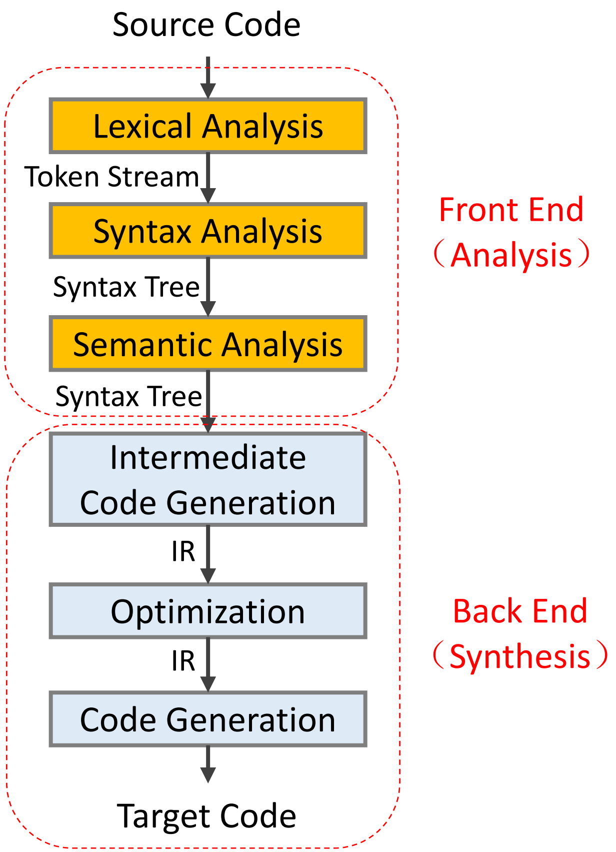
第12讲：语义分析(1)

张献伟

xianweiz.github.io

DCS290, 3/31/2022

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?

x#y = 1

x = 1 y = 2

int x; y = x(1)