Compilation Principle
编 译 原 理

第5讲：语法分析(2)

张献伟

xianweiz.github.io

DCS290, 3/9/2023

Review Questions

- Q1: RE to describe L={$a^n c b^n$}, where $0 \le n \le 5$?

  Yes. RE=acb|aacbb|aaacbbb|aaaacbbbb|aaaaacbbbbb

- Q2: is RL applicable to syntax analysis? Why?

  No. RL is not powerful enough, e.g., no nested structure

- Q3: input and output of parser?

  Input: tokens from lexer; output: parse tree or AST

- Q4: how does grammar relate to syntax?

  Grammar is used to specify syntax.

- Q5: productions of grammar?

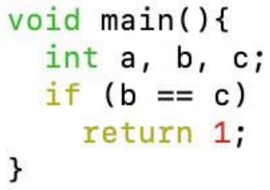
  LHS → *RHS*. e.g., E → E + E

Syntax Analysis[语法分析]

- Informal description of variable declarations in C[变量声明]
  - Starts with *int* or *float* as the first token[类型]
  - Followed by one or more *identifier* tokens, separated by token *comma*[逗号分隔的标识符]
  - Followed by token *semicolon*[分号]
- To *check whether a program is well-formed* requires a specification of *what is a well-formed program*[语法定义]
  - The specification be precise[正确]
  - The specification be complete[完备]
    - ▫ Must cover all the syntactic details of the language
  - The specification must be convenient[便捷] to use by both language designer and the implementer
- A **context free grammar** meets these requirements

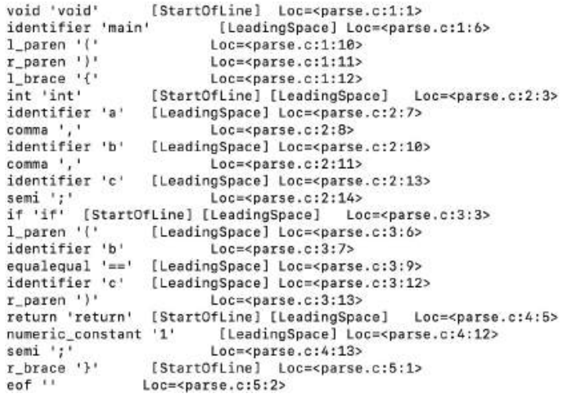
https://www.cse.iitb.ac.in/~uday/courses/cs324-07/syntax-analysis.pdf

Example



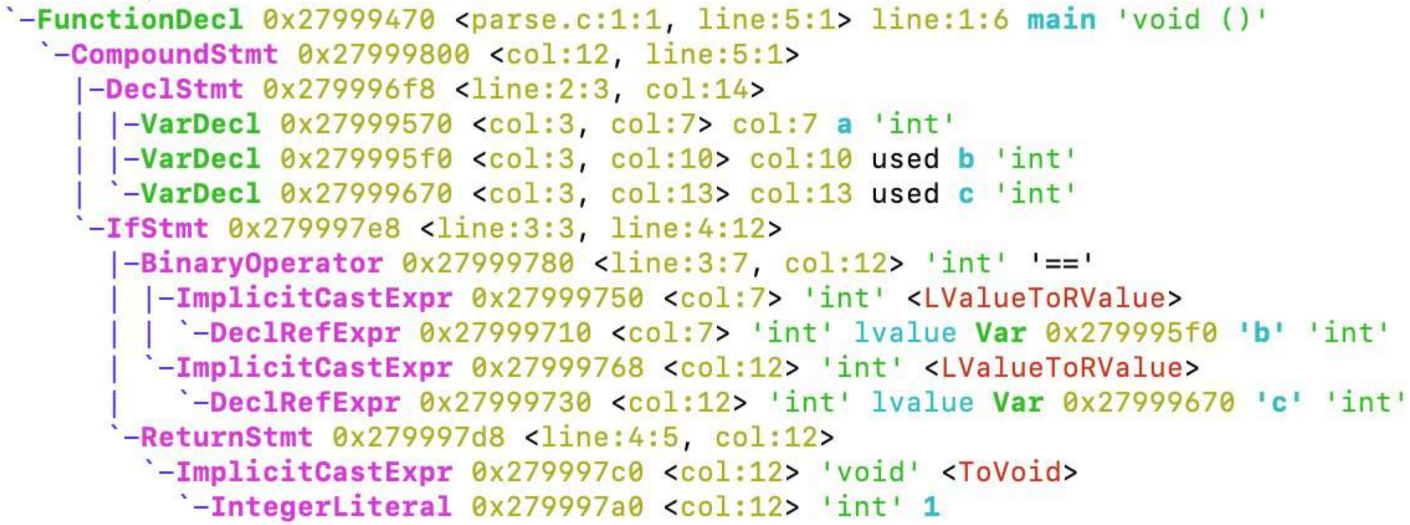
```
void main(){
  int a, b, c;
  if (b == c)
    return 1;
}
```

$clang -cc1 -dump-tokens test.c

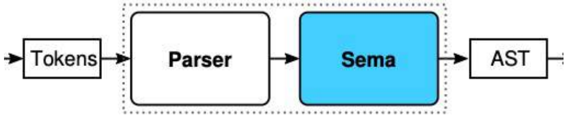


```
void 'void'      [StartOfLine]  Loc=<parse.c:1:1>
identifier 'main'        [LeadingSpace] Loc=<parse.c:1:6>
l_paren '('             Loc=<parse.c:1:10>
r_paren ')'             Loc=<parse.c:1:11>
l_brace '{'             Loc=<parse.c:1:12>
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<parse.c:2:3>
identifier 'a'   [LeadingSpace] Loc=<parse.c:2:7>
comma ','               Loc=<parse.c:2:8>
identifier 'b'   [LeadingSpace] Loc=<parse.c:2:10>
comma ','               Loc=<parse.c:2:11>
identifier 'c'   [LeadingSpace] Loc=<parse.c:2:13>
semi ';'                Loc=<parse.c:2:14>
if 'if'  [StartOfLine] [LeadingSpace]   Loc=<parse.c:3:3>
l_paren '('      [LeadingSpace] Loc=<parse.c:3:6>
identifier 'b'          Loc=<parse.c:3:7>
equalequal '=='  [LeadingSpace] Loc=<parse.c:3:9>
identifier 'c'   [LeadingSpace] Loc=<parse.c:3:12>
r_paren ')'             Loc=<parse.c:3:13>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<parse.c:4:5>
numeric_constant '1'     [LeadingSpace] Loc=<parse.c:4:12>
semi ';'                Loc=<parse.c:4:13>
r_brace '}'      [StartOfLine]  Loc=<parse.c:5:1>
eof ''          Loc=<parse.c:5:2>
```

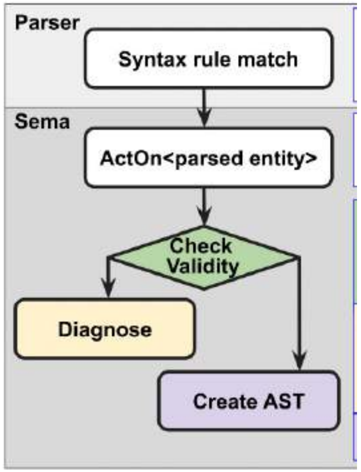
$clang -Xclang -ast-dump -fsyntax-only test.c



```
`-FunctionDecl 0x27999470 <parse.c:1:1, line:5:1> line:1:6 main 'void ()'
  `-CompoundStmt 0x27999800 <col:12, line:5:1>
    |-DeclStmt 0x279996f8 <line:2:3, col:14>
    | |-VarDecl 0x27999570 <col:3, col:7> col:7 a 'int'
    | |-VarDecl 0x279995f0 <col:3, col:10> col:10 used b 'int'
    | `-VarDecl 0x27999670 <col:3, col:13> col:13 used c 'int'
    `-IfStmt 0x279997e8 <line:3:3, line:4:12>
      |-BinaryOperator 0x27999780 <line:3:7, col:12> 'int' '=='
      | |-ImplicitCastExpr 0x27999750 <col:7> 'int' <LValueToRValue>
      | | `-DeclRefExpr 0x27999710 <col:7> 'int' lvalue Var 0x279995f0 'b' 'int'
      | `-ImplicitCastExpr 0x27999768 <col:12> 'int' <LValueToRValue>
      |   `-DeclRefExpr 0x27999730 <col:12> 'int' lvalue Var 0x27999670 'c' 'int'
      `-ReturnStmt 0x279997d8 <line:4:5, col:12>
        `-ImplicitCastExpr 0x279997c0 <col:12> 'void' <ToVoid>
          `-IntegerLiteral 0x279997a0 <col:12> 'int' 1
```

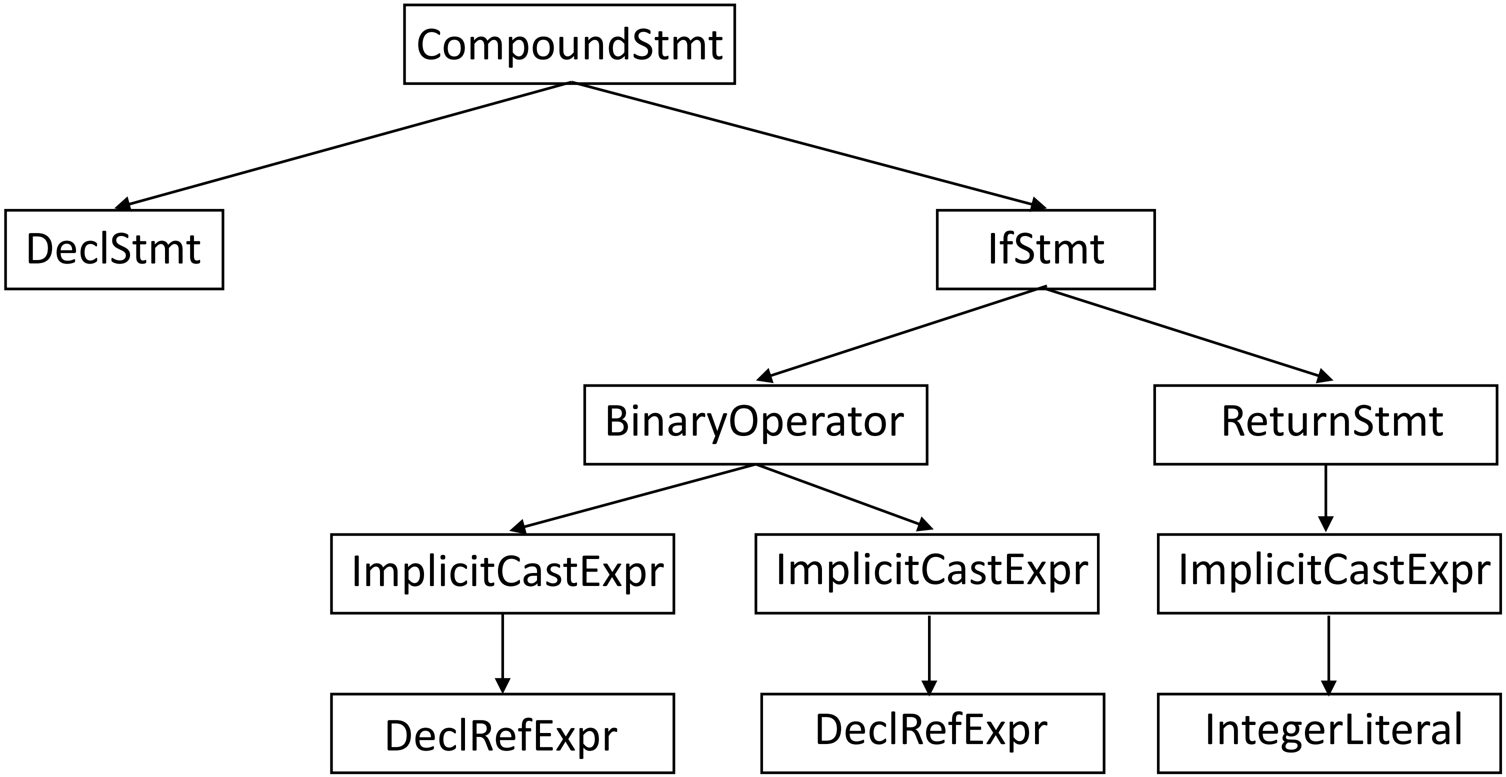


Sema is tight coupling with parser



https://llvm.org/devmtg/2019-10/slides/ClangTutorial-Stulova-vanHaastregt.pdf

Example (cont.)


CompoundStmt

DeclStmt

IfStmt

BinaryOperator

ReturnStmt

ImplicitCastExpr

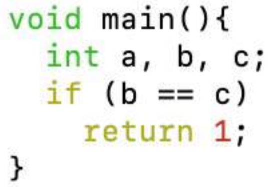
ImplicitCastExpr

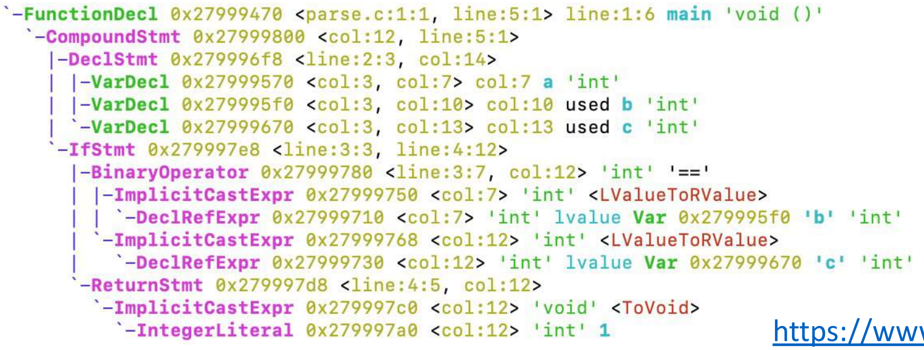
ImplicitCastExpr

DeclRefExpr

DeclRefExpr

IntegerLiteral

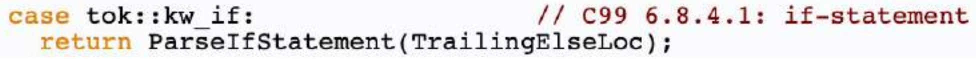

```c
void main(){
  int a, b, c;
  if (b == c)
    return 1;
}
```


```
`-FunctionDecl 0x27999470 <parse.c:1:1, line:5:1> line:1:6 main 'void ()'
  `-CompoundStmt 0x27999800 <col:12, line:5:1>
    |-DeclStmt 0x279996f8 <line:2:3, col:14>
    | |-VarDecl 0x27999570 <col:3, col:7> col:7 a 'int'
    | |-VarDecl 0x279995f0 <col:3, col:10> col:10 used b 'int'
    | `-VarDecl 0x27999670 <col:3, col:13> col:13 used c 'int'
    `-IfStmt 0x279997e8 <line:3:3, line:4:12>
      |-BinaryOperator 0x27999780 <line:3:7, col:12> 'int' '=='
      | |-ImplicitCastExpr 0x27999750 <col:7> 'int' <LValueToRValue>
      | | `-DeclRefExpr 0x27999710 <col:7> 'int' lvalue Var 0x279995f0 'b' 'int'
      | `-ImplicitCastExpr 0x27999768 <col:12> 'int' <LValueToRValue>
      |   `-DeclRefExpr 0x27999730 <col:12> 'int' lvalue Var 0x27999670 'c' 'int'
      `-ReturnStmt 0x279997d8 <line:4:5, col:12>
        `-ImplicitCastExpr 0x279997c0 <col:12> 'void' <ToVoid>
          `-IntegerLiteral 0x279997a0 <col:12> 'int' 1
```

https://www.cnblogs.com/jourluohua/p/14524955.html

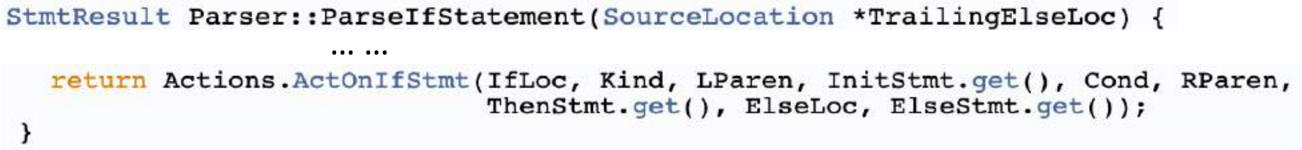
Example (cont.)

https://clang.llvm.org/doxygen/ParseStmt_8cpp_source.html



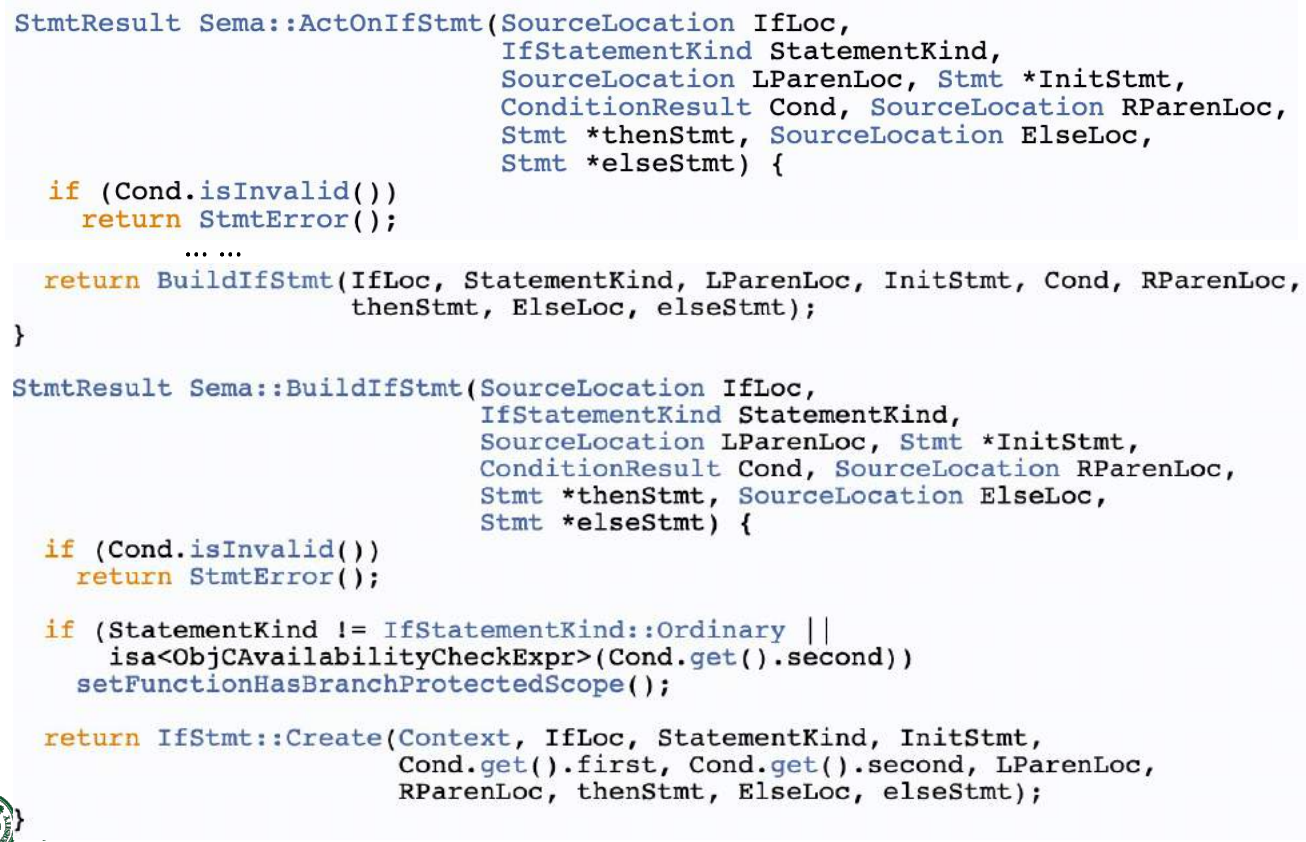
```
case tok::kw_if:                    // C99 6.8.4.1: if-statement
  return ParseIfStatement(TrailingElseLoc);
```

… …



```
StmtResult Parser::ParseIfStatement(SourceLocation *TrailingElseLoc) {
                        … …
   return Actions.ActOnIfStmt(IfLoc, Kind, LParen, InitStmt.get(), Cond, RParen,
                              ThenStmt.get(), ElseLoc, ElseStmt.get());
 }
```

https://clang.llvm.org/doxygen/SemaStmt_8cpp_source.html

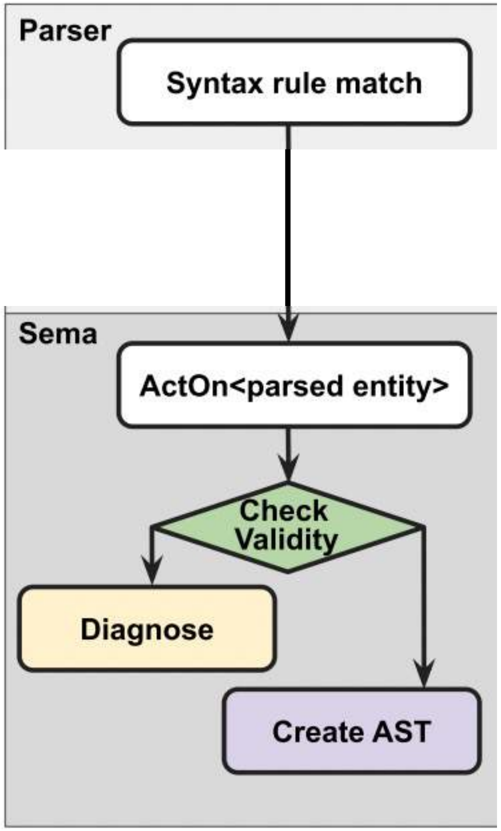


```
StmtResult Sema::ActOnIfStmt(SourceLocation IfLoc,
                             IfStatementKind StatementKind,
                             SourceLocation LParenLoc, Stmt *InitStmt,
                             ConditionResult Cond, SourceLocation RParenLoc,
                             Stmt *thenStmt, SourceLocation ElseLoc,
                             Stmt *elseStmt) {
  if (Cond.isInvalid())
    return StmtError();
             … …
  return BuildIfStmt(IfLoc, StatementKind, LParenLoc, InitStmt, Cond, RParenLoc,
                     thenStmt, ElseLoc, elseStmt);
}

StmtResult Sema::BuildIfStmt(SourceLocation IfLoc,
                             IfStatementKind StatementKind,
                             SourceLocation LParenLoc, Stmt *InitStmt,
                             ConditionResult Cond, SourceLocation RParenLoc,
                             Stmt *thenStmt, SourceLocation ElseLoc,
                             Stmt *elseStmt) {
  if (Cond.isInvalid())
    return StmtError();

  if (StatementKind != IfStatementKind::Ordinary ||
      isa<ObjCAvailabilityCheckExpr>(Cond.get().second))
    setFunctionHasBranchProtectedScope();

  return IfStmt::Create(Context, IfLoc, StatementKind, InitStmt,
                        Cond.get().first, Cond.get().second, LParenLoc,
                        RParenLoc, thenStmt, ElseLoc, elseStmt);
}
```



Context Free Grammar[上下文无关文法]

- Formal definition[形式化定义]: 4 components **{T, N, s, δ}**
  - *T* is a finite set of terminals (i.e., token names from lexer)
  - *N* is a finite set of non-terminals
    - syntactic variables denoting sets of strings, helpful for defining language generated from the grammar
  - *S* is a special nonterminal (from *N*) called the start symbol
  - δ is a finite set of production rules of the form such as A→ α, where A is from *N* and α from (*N* ∪ *T*)∗
- CFG of variable declarations
  - $\{\underbrace{\{\text{id , int float ;}\}}_{T}, \underbrace{\{\textit{declaration type idlist}\}}_{N}, \underbrace{\textit{declaration}}_{s}, \delta\}$
- Production rules (δ)

declaration → *type idlist ;*

idlist → id | *idlist* , id

type → int | float

```
void main() {
 int;
 int a,;
 int b, c;
}
```

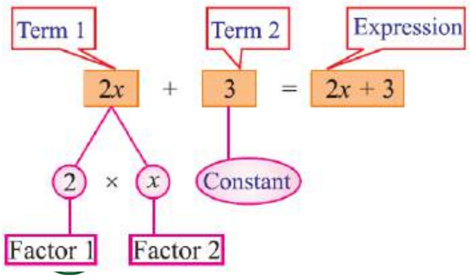
https://www.cse.iitb.ac.in/~uday/courses/cs324-07/syntax-analysis.pdf

Notational Conventions[标识规范]

- These symbols are terminals[终结符]
  - Lowercase letters early in the alphabet, e.g., a, b, c[靠前小写字母]
  - Operator symbols such as +, *, …[运算符]
  - Punctuation symbols such as (, , …[标点符号]
  - Digits 0, 1, …, 9[数字]
  - Boldface strings such as **id** or **if**, each is a single terminal symbol
- These symbols are non-terminals[非终结符]
  - Uppercase letters early in alphabet, e.g., A, B, C[靠前大写字母]
  - The letter S, which, when it appears, is usually the start symbol
  - *Lowercase, italic* names such as *expr* or *stmt*[小写斜体]
  - When discussing programming constructs, uppercase letters may represent non-terminals for the constructs
    - E.g., *E*: expression[表达式], T: term[项], F: factor[因子]

Notational Conventions (cont.)

- Uppercase letters late in alphabet, e.g., X, Y, Z, represent grammar symbols
  - Either non-terminals or terminals
- Lowercase letters late in alphabet, chiefly u, v, …, z, represent (possibly empty) strings of terminals
- Lowercase Greek letters, e.g.,α, β, γ represent (possibly empty) strings of grammar symbols
  - $A \rightarrow \alpha$
- Unless stated otherwise, the head of the first production is the start symbol[开始符号]



$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid \textbf{id}$

Start symbol: E

Nonterminals: E, T and F

Terminals: everything else

Production Rule and Derivation[推导]

- **Production rule**[产生规则]: *LHS* → *RHS*
  - Aliases[别名]: *LHS* ≡ head, *RHS* ≡ body
  - Meaning[含义]: *LHS* can be constructed (or replaced) with *RHS*
- **Derivation**[推导]: a series of applications of production rules
  - Replace a non-terminal by the corresponding *RHS* of a production
- $\beta \Rightarrow \alpha$
  - Meaning: string α is derived from β
  - $\beta \Rightarrow \alpha$: derives in one step
  - $\beta \Rightarrow^* \alpha$: derives in zero or more steps
  - $\beta \Rightarrow^+ \alpha$: derives in one or more steps
- Example: $A \Rightarrow 0A \Rightarrow 00B \Rightarrow 000$
  - $A \Rightarrow^* 000$
  - $A \Rightarrow^+ 000$

Derivation[推导]

- If $S \Rightarrow^* \alpha$, where S is the start symbol of grammar G

- α: **sentential form** of G[句型]
  - A sentential form may contain both terminals and non-terminals (and can be empty)

S = subject, V = verb, O = object
SV: She laughed.
SVO: She opened the door.

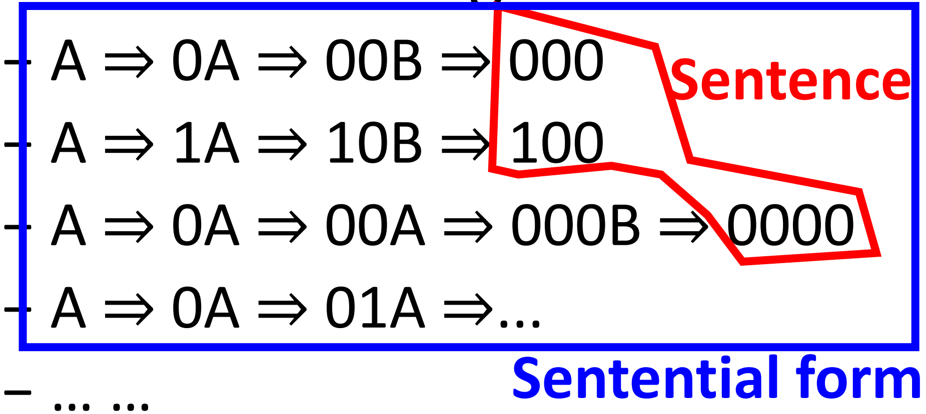
- α: **sentence** of G[句子]
  - A sentential form with no non-terminals[仅包含终结符]

- **Language**[语言] generated by a grammar
  - $L(G) = \{w: S \Rightarrow {}^*w, w \in V_T{}^* \}$
  - A string of terminal *w* is in L(G), **iff** *w* is a sentence of G (or $S \Rightarrow^* w$)

Example

- Grammar $G = \{T, N, s, \delta\}$
  - $T = \{0, 1\}$
  - $N = \{A, B\}$
  - $s = A$
  - $\delta = \{ A \rightarrow 0A \mid 1A \mid 0B,\ B \rightarrow 0 \}$

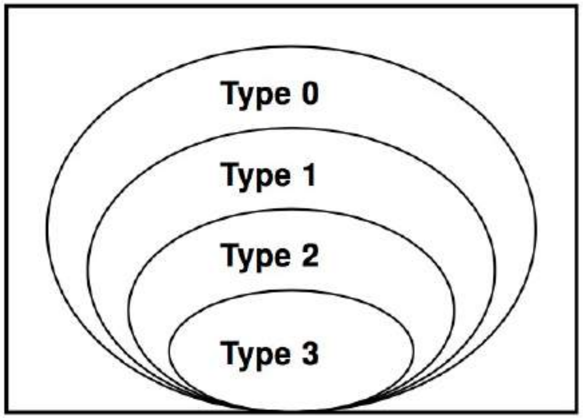
- Derivation: from grammar to language[文法到语言]



  - $A \Rightarrow 0A \Rightarrow 00B \Rightarrow 000$
  - $A \Rightarrow 1A \Rightarrow 10B \Rightarrow 100$
  - $A \Rightarrow 0A \Rightarrow 00A \Rightarrow 000B \Rightarrow 0000$
  - $A \Rightarrow 0A \Rightarrow 01A \Rightarrow \ldots$
  - … …

Language Classification: Chomsky

- **Language classification** based on form of grammar rules
- Four types of grammars:
  - Type 0 — unrestricted grammar
    - 0型文法 – 无限制文法
  - Type 1 — context sensitive grammar(CSG)
    - 1型文法 – 上下文有关文法
  - Type 2 — context free grammar (CFG)
    - 2型文法 – 上下文无关文法
  - Type 3 — regular grammar
    - 3型文法 – 正则文法



- Regular Grammar ⊆ CFG ⊆ CSG ⊆ Unrestricted Grammar

Chomsky hierarchy

In 1957, Noam Chomsky published *Syntactic Structures*, an landmark book that defined the so-called Chomsky hierarchy of languages

American linguist, philosopher, cognitive scientist, historian, and activist.

His work has influenced fields such as computer science, mathmatics and psychology.

Type 0: Unrestricted Grammar

- Form of rules α→β
  - where $\alpha \in (N \cup T)^+$, $\beta \in (N \cup T)^*$
- Implied restrictions
  - LHS: no ε allowed
- Example:
  - aB → aCD: LHS is shorter than RHS
  - aAB → aB : LHS is longer than RHS
  - A → ε: ε-productions are allowed
- Derivations
  - Derivation strings may contract and expand repeatedly (since LHS may be longer or shorter than RHS)
  - Unbounded number of productions before target string

Type 1: Context Sensitive Grammar

- Form of rules: $\alpha A \beta \rightarrow \alpha\gamma\beta$
  - where $A \in N$, $\alpha, \beta \in (N \cup T)^*$, $\gamma \in (N \cup T)^+$
- Replace *A* by γ only if found <u>in the context of α and β</u>
- Implied restrictions
  - LHS: shorter or equal to RHS
  - RHS: no ε allowed
- Example:
  - aAB→aCB: replace A with C when in between a and B
  - A → C: replace A with C regardless of context
- Derivations
  - Derivation strings may only expand
  - Bounded number of derivations before target string

Type 2: Context Free Grammar

- Form of rules: $A \to \gamma$
  - where $A \in N$, $\gamma \in (N \cup T)^+$
- Replace A by γ (no context can be specified)
- Implied restrictions
  - LHS: a single non-terminal
  - RHS: no ε allowed
    - Sometimes relaxed to simplify grammar but rules can always be rewritten to exclude ε-productions
- Example:
  - A → aBc: replace A with aBc regardless of context

> $L = \{ a^n b^n \mid n \geq 0 \}$ is **NOT regular** but **IS** a **context-free language**.
>
> For the following CFG $G = < T, N, S, \delta >$ generates L:
> $T = \{ a, b \}$, $N = \{ S \}$ and $\delta = \{ S \text{ -> } aSb , S \text{ -> } ab \}$

Type 3: Regular Grammar

- Form of rules A→α,or A→αB
  - where A,B ∈ N, α ∈ T
- In terms of FA:
  - Move from state A to state B on input α
- Implied restrictions
  - LHS: a single non-terminal
  - RHS: a terminal or a terminal followed by a non-terminal
- Example: A → 1A | 0
  - RE: **1*0**
- Derivation:
  - Derivation string length increases by 1 at each step

In Practice[实际中]

- Every regular language is a context-free language
  - Context-free languages more general than regular languages
- If PLs are context-sensitive, why use CFGs for parsing?
  - Perfectly suited to describing recursive syntax of expressions and statements
  - CSG parsers are provably inefficient
  - Most PL constructs are context-free
    - if-stmt, declarations
  - The remaining context-sensitive constructs can be analyzed at the semantic analysis stage
    - e.g. def-before-use, matching formal/actual parameters
- In PLs
  - Regular language for lexical analysis
  - Context-free language for syntax analysis

Grammar and Derivation[文法与推导]

- **Grammar** is used to derive string or construct parser[文法]
- A **derivation** is a sequence of applications of rules[推导]
  - Starting from the start symbol
  - S⇒... ⇒... ⇒... ⇒ (sentence)
  - There are choices at each sentential form
    - choice of the nonterminal to be replaced[替换哪个？]
    - choice of a rule corresponding to the nonterminal[怎么替换？]
- Instead of choosing the nonterminal to be replaced, in an *arbitrary* fashion, it is possible to make an *uniform* choice at each step[统一化选择替换哪个]
- **Leftmost** and **Rightmost** derivations[最左和最右推导]
  - At each derivation step, leftmost derivation always replaces the leftmost non-terminal symbol
  - Rightmost derivation always replaces the rightmost one

https://www.cse.iitb.ac.in/~uday/courses/cs324-07/syntax-analysis.pdf

Example

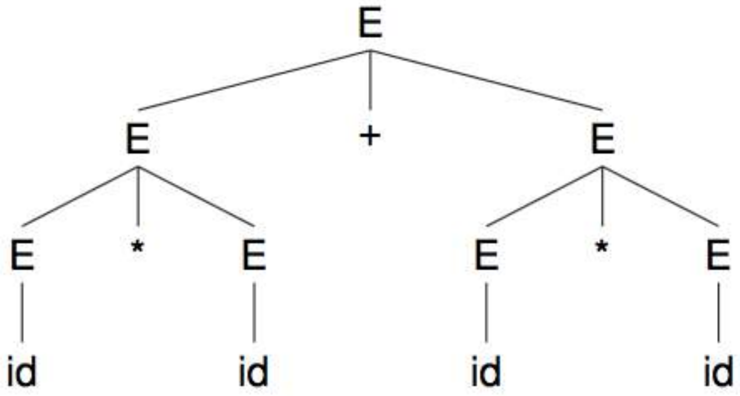
- Two derivations of string “id * id + id * id” using grammar:
  $E \to E*E \mid E+E \mid (E) \mid \mathrm{id}$

- Leftmost derivation[最左推导]
  - $E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow \mathrm{id} * E + E \Rightarrow \mathrm{id} * \mathrm{id} + E \Rightarrow \ldots \Rightarrow \mathrm{id} * \mathrm{id} + \mathrm{id} * \mathrm{id}$
- Rightmost derivation[最右推导]
  - $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * \mathrm{id} \Rightarrow E + \mathrm{id} * \mathrm{id} \Rightarrow \ldots \Rightarrow \mathrm{id} * \mathrm{id} + \mathrm{id} * \mathrm{id}$

- Derivations can be summarized as a parse tree[分析树]

Parse Trees[分析树]

- Both previous derivations result in the same parse tree:



Two derivations of string
"id * id + id * id"
using grammar:
E→*E***E* | *E*+*E* | (*E*) | id

- A **parse tree** is a graphical representation of a derivation
  - But filters out the order in which productions are applied to replace non-terminals[过滤了推导顺序信息]
  - Each interior node represents the application of a production
    - Labeled with the non-terminal in the LHS of production
  - Leaves are labeled by terminals or non-terminals
    - Constitutes a sentential form (read from left to right)
    - Called the ***yield[产出]*** or ***frontier[边缘]*** of the tree

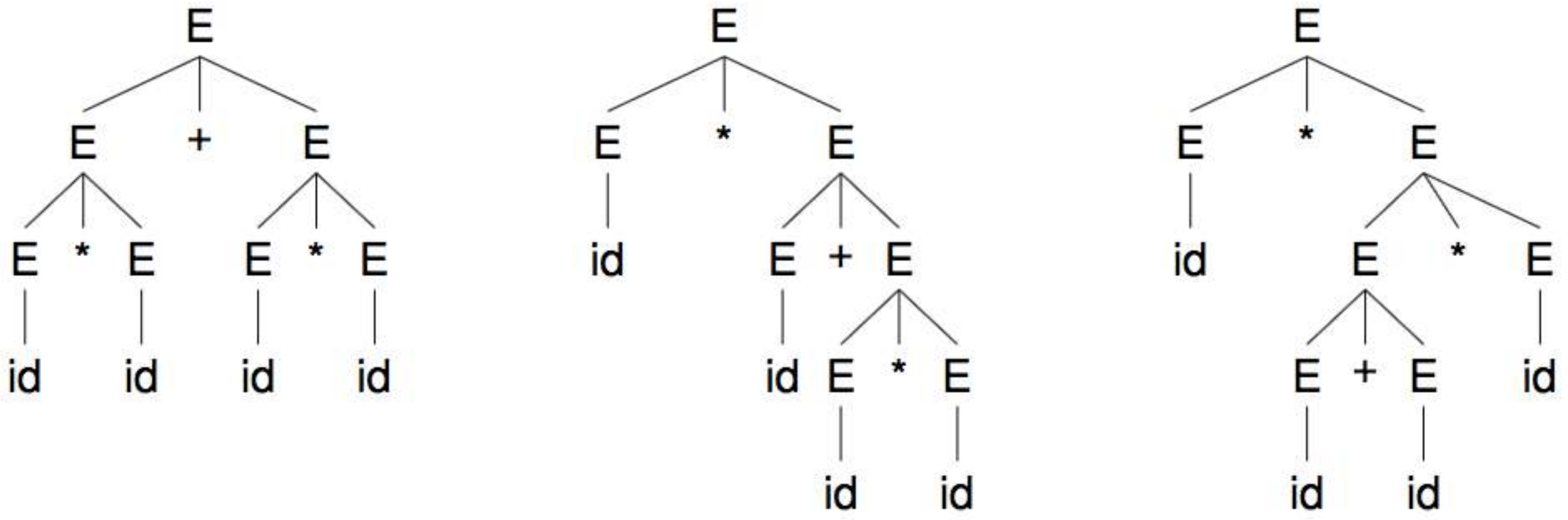
Parse Trees (cont.)

- Derivations and parse trees: many-to-one relationship
  - Leftmost derivation order: builds tree left to right
  - Rightmost derivation order: builds tree right to left
  - Different parser implementations choose different orders
  - One-to-one relationships between parse trees and either leftmost or rightmost derivations[最左或最右推导与分析树具有一对一对应关系]

- Program structure does not depend on order of rule application, instead it depends on what production rules are applied
  - Grammar must define unambiguously set of rules applied

Different Parse Trees

- Grammar *E*→*E***E* | *E*+*E* | (*E*) | id is ambiguous[二义的]
  - String id * id + id * id can result in 3 parse trees (and more)



The deepest sub-tree is traversed first, thus highest precedence

- Grammar can apply different rules to derive same string
  - Meaning of parse tree 1: (id * id) + (id * id)
  - Meaning of parse tree 2: id * (id + (id * id))
  - Meaning of parse tree 3: id * ((id + id) * id)

Preorder?

Inorder? ✓

Postorder?

Ambiguity[二义性]

- Grammar G is ambiguous if
  - It produces more than one parse tree for some sentence
  - i.e., there exist a string *str* ∈ L(G) such that
  - more than one parse tree derives *str*
    ≡ more than one leftmost derivation derives *str*
    ≡ more than one rightmost derivation derives *str*
- Unambiguous grammars are preferred for most parsers[文法最好没有歧义性]
  - Ambiguity of the grammar implies that at least some strings in its language have different structures (parse trees)
  - Thus, such a grammar is unlikely to be useful for a programming language, because two structures for the same string (program) implies two different meanings (executable equivalent programs) for this program

http://infolab.stanford.edu/~ullman/ialc/slides/slides7.pdf