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5讲：语法分析(1)

张献伟

xianweiz.github.io

DCS290, 3/14/2024

Questions

Handwritten, or email to
chenhq79@mail2.sysu.edu.cn

QUIZ

- Q1: write RE for 2n and 3n *a*s (*aa*, *aaa*, …), including ε?

  RE= (aa)* | (aaa)*

- Q2: lexical analysis of 'if x*%5'?

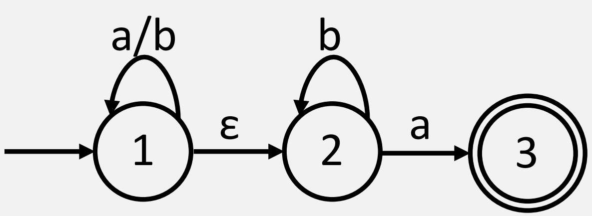
  (keyword, if), (id, x), (sym, *), (sym, %), (num, 5)

- Q3: regard lexer implementation, why NFA → DFA?

  Trade-off space for speed; DFA is more efficient

- Q4: RE of the FA?

  (a|b)*b*a



- Q5: start state of the equivalent DFA?

  ε-closure(1) = {1, 2}

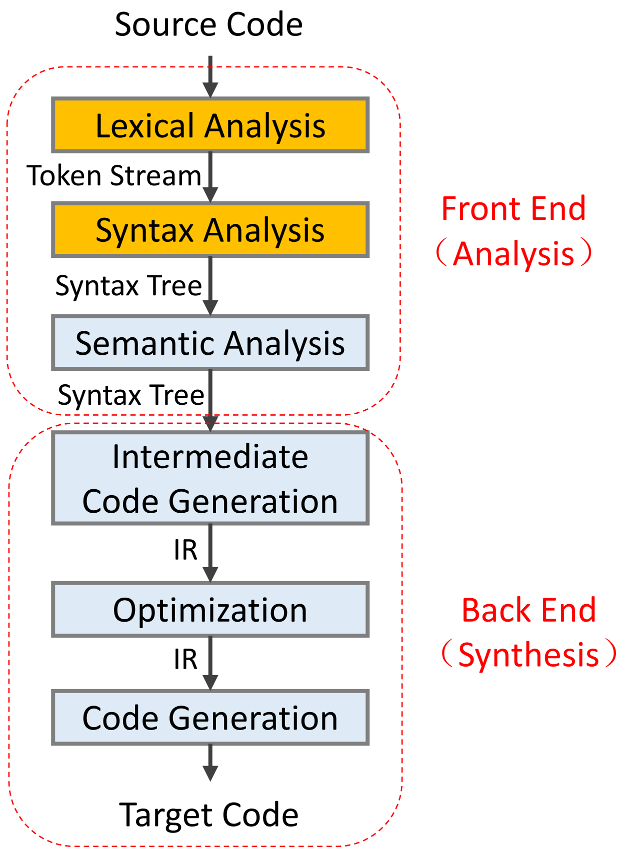
  ε-closure(move({12}, a)) = ε-closure({1,3}) ⟹ {1, 2, 3}

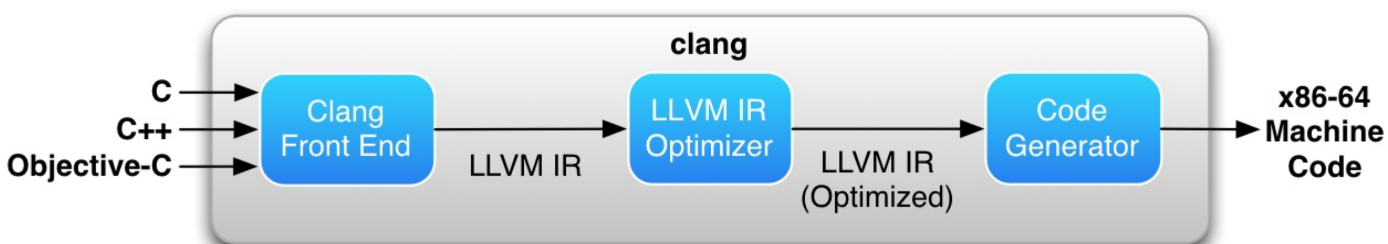
  ε-closure(move({12}, b)) = ε-closure({1,2}) ⟹ {1, 2}

Beyond Regular Languages

- Regular languages are expressive enough for tokens
  - Can express identifiers, strings, comments, etc.
- However, it is the weakest (least expressive) language
  - Many languages are not regular
  - C programming language is not
    - The language matching braces "{{{...}}}" is also not
  - FA cannot count # of times char encountered
    - $L = \{a^n b^n \mid n \geq 1\}$
    - Crucial for analyzing languages with nested structures (e.g. nested for loop in C language)
- We need a more powerful language for parsing
  - Later, we will discuss context-free languages (CFGs)

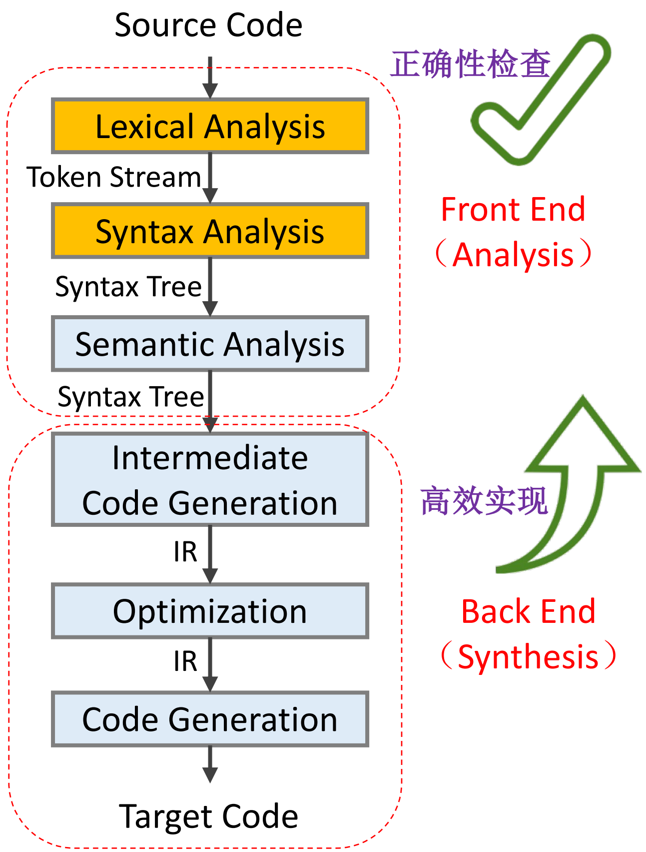
Compilation Phases[编译阶段]

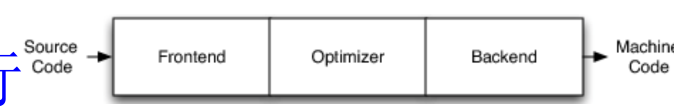




Compilation Procedure[编译过程]

- **前端**（分析）：对源程序，识别语法结构信息，理解语义信息，反馈出错信息
  - 词法分析（Lexical Analysis）词
  - 语法分析（Syntax Analysis） 语句
  - 语义分析（Semantic Analysis） 上下文
- **后端**（综合）：综合分析结果，生成语义上等价于源程序的目标程序
  - 中间代码生成（Intermediate Code Generation）
    - Intermediate representation (IR)转换
  - 代码优化（Code Optimization） 更好
  - 目标代码生成（Code Generation）可执行





Example

- $vim test.c

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

- $clang -cc1 -dump-tokens ./test.c
- $clang -o test test.c

```
test.c:1:1: warning: return type of 'main' is not 'int' [-Wmain-return
void main() {
^
test.c:1:1: note: change return type to 'int'
void main() {
^~~~
int
test.c:2:3: warning: declaration does not declare anything [-Wmissing-
ns]
  int;
  ^~~
test.c:3:9: error: expected identifier or '('
  int a,;
        ^
2 warnings and 1 error generated.
```

```
void 'void'
identifier 'main'
l_paren '('
r_paren ')'
l_brace '{'
int 'int'
semi ';'
int 'int'
identifier 'a'
comma ','
semi ';'
int 'int'
identifier 'b'
comma ','
identifier 'c'
semi ';'
r_brace '}'
eof ''
```

Example

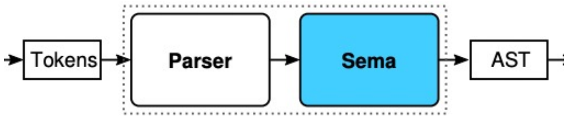
```c
void main(){
  int a, b, c;
  if (b == c)
    return 1;
}
```

$clang -cc1 -dump-tokens test.c

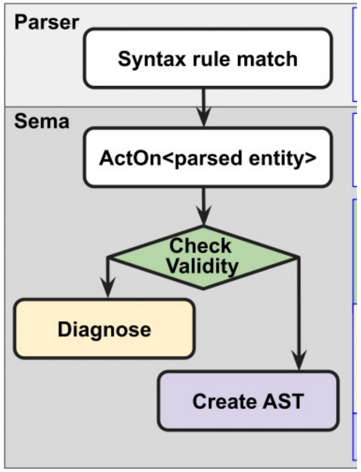
```
void 'void'      [StartOfLine]  Loc=<parse.c:1:1>
identifier 'main'        [LeadingSpace] Loc=<parse.c:1:6>
l_paren '('             Loc=<parse.c:1:10>
r_paren ')'             Loc=<parse.c:1:11>
l_brace '{'             Loc=<parse.c:1:12>
int 'int'        [StartOfLine] [LeadingSpace]   Loc=<parse.c:2:3>
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
```



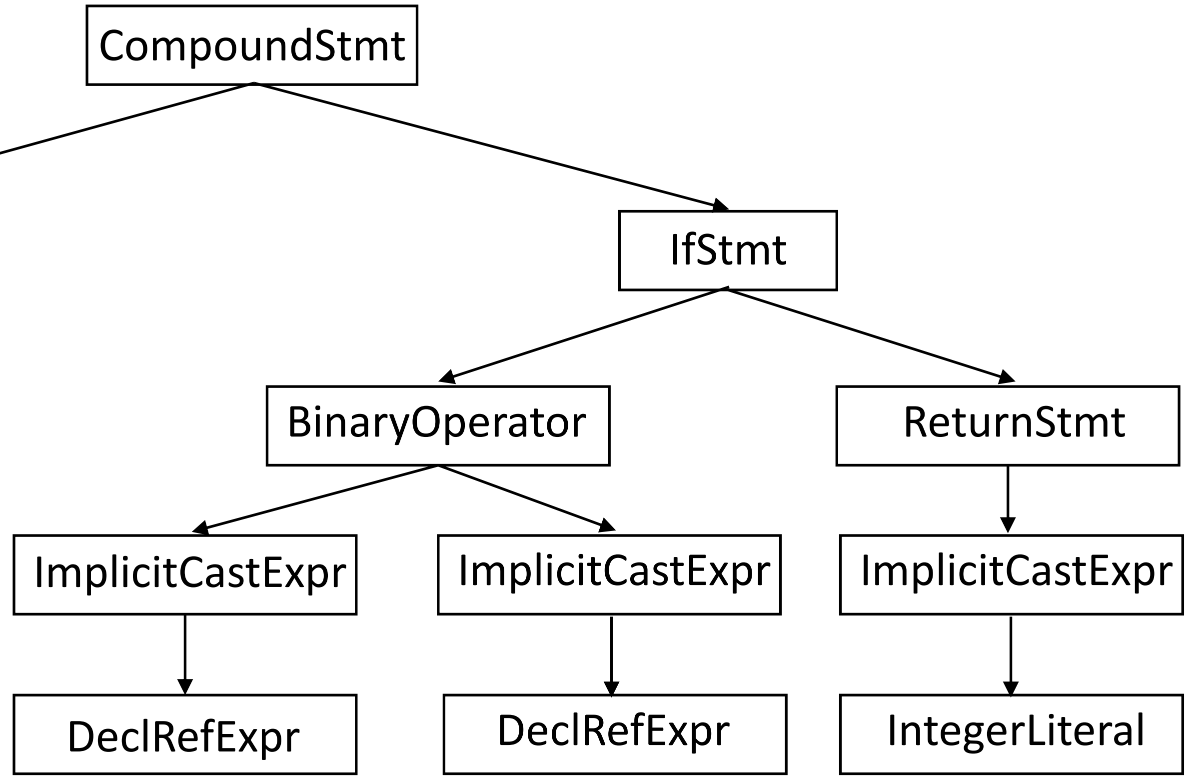
Sema is tight coupling with parser



SUN YAT-SEN UNIVERSITY 中山大學

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

Example (cont.)




```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());
 }
```

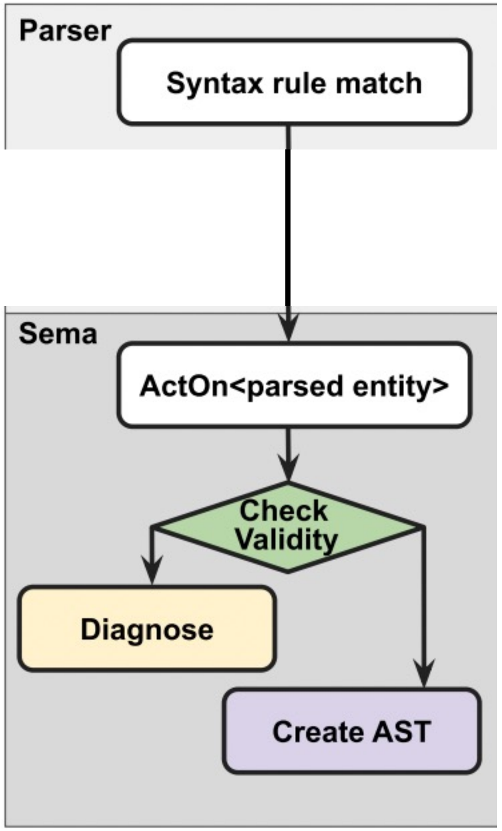
https://clang.llvm.org/doxygen/SemaStmt_8cpp_source.html

```
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);
}
```



Syntax Analysis[语法分析]

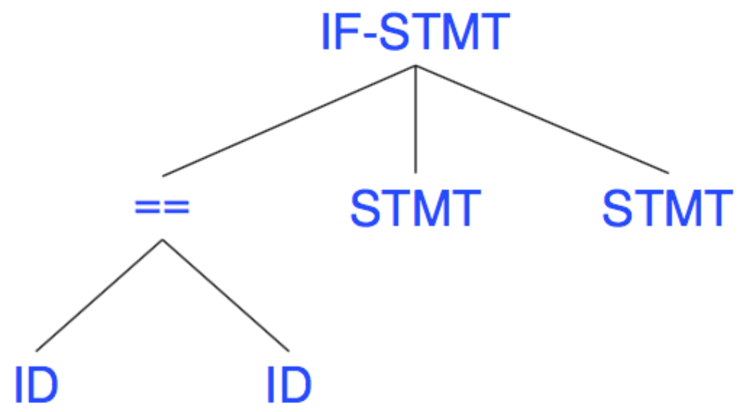
- Second phase of compilation[第二阶段]
  - Also called as **parser**
- Parser obtains a string of tokens from the lexical analyzer[以token作为输入]
  - **Lexical analyzer** reads the chars of the source program, groups them into lexically meaningful units called **lexemes**
  - and produces as output **tokens** representing these lexemes
    - Token: <token name, attribute value>
  - Token names are used by parser for syntax analysis
    - tokens → parse tree/AST
- **Parse tree**[分析树]
  - Graphically represent the syntax structure of the token stream

Parsing Example

- Input: if(x==y) … else …[源程序输入]
- Parser input (Lexical output)[语法分析输入]

  KEY(IF) SYM(‘(’) ID(x) OP(‘==’) ID(y) SYM(‘)’) … KEY(ELSE) …

- Parser output[语法分析输出]



Parsing Example (cont.)

- Example: <id, x> <op, *> <op, %>
  - Is it a valid token stream in C language? **YES**
  - Is it a valid statement in C language (x *%)? **NO**

- Not every sequence of tokens are valid
  - Parser must distinguish between valid and invalid token sequence

- We need a method to describe what is valid sequence?
  - To specify the syntax of a programming language
  - RE cannot be used

How to Specify Syntax?

- How can we specify a syntax with nested structures?
  - Is it possible to use RE/FA?
  - L(Regular Expression) ≡ L(Finite Automata)

RE to describe $L=\{a^n cb^n\}$, where $0 \le n \le 4$?

RE=c|acb|aacbb|aaacbbb|aaaacbbbb

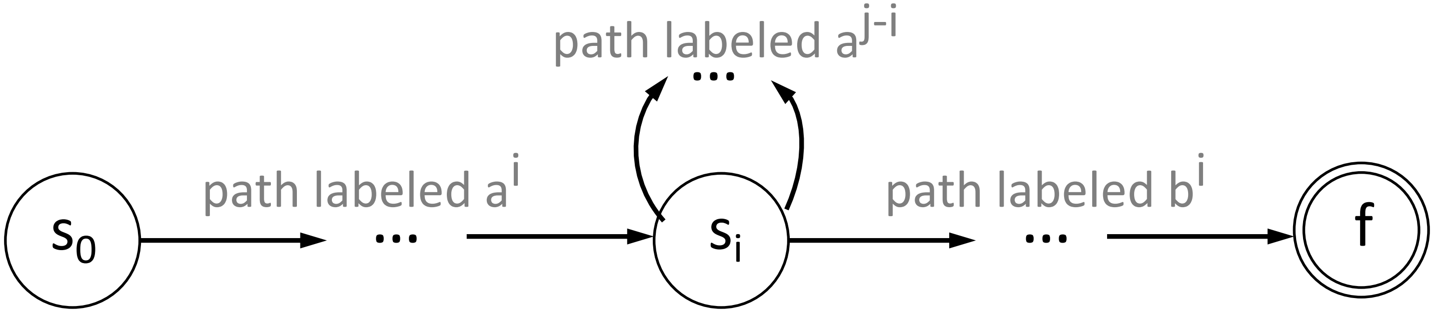
- RE/FA is not powerful enough
  - $L = \{a^n b^n \mid n \geq 1\}$ is not a Regular Language

- Example: matching parenthesis: # of '(' == # of ')'
  - (x+y)*z ✓
  - ((x+y)+y)*z ✓
  - (...(((x+y)+y)+y)...) ✓
  - ((x+y)+y)+y)*z ✗

SUN YAT-SEN UNIVERSITY

RE/FA is NOT Powerful Enough

- $L = \{a^n b^n \mid n \geq 1\}$ is NOT a Regular Language
  - Suppose L were the language defined by regular expression
  - Then we could construct a DFA D with k states to accept L
  - Since D has only k states, for an input beginning with more than k a's, D must enter some state twice, say s_i
  - Suppose that the path from s_i back to itself is labeled with a^{j-i}
  - Since $a^i b^i$ is in L, there must be a path labeled b^i from s_i to an accepting state f
  - But, there is also a path from s_0 through s_i to f labelled $a^j b^i$
  - Thus, D also accepts $a^j b^i$, which is not in L, contradicting the assumption that L is the language accepted by D



RE/FA is NOT Powerful Enough(cont.)

- $L = \{a^n b^n \mid n \geq 1\}$ is not a Regular Language
  - Proof → Pumping Lemma (泵引理)
  - FA does not have any memory (FA cannot count)
    - The above *L* requires to keep count of a's before seeing b's

- Matching parenthesis is not a RL
- Any language with nested structure is not a RL
  - if … if … else … else
- Regular Languages
  - Weakest formal languages that are widely used
  - Simple yet powerful (able to express patterns)

https://shirts.ly/ok-fine-whatever

What Language Do We Need?

- C-language syntax: **Context Free Language** (CFL)[上下文无关语言]   e.g., ‘else’ is always ‘else’, wherever you place it
  - A broader category of languages that includes languages with nested structures
- Before discussing CFL, we need to learn a more general way of specifying languages than RE, called **Grammars**[文法]
  - Can specify both RL and CFL
  - and more ...
- Everything that can be described by a regular expression can also be described by a grammar
  - Grammars are most useful for describing nested structures

Concepts

- **Language**[语言]
  - Set of strings over alphabet
    - *String*: finite sequence of symbols
    - *Alphabet*: finite set of symbols

- **Grammar**[文法]
  - To systematically describe the syntax of programming language constructs like expressions and statements

- **Syntax**[语法]
  - Describes the proper form of the programs
  - Specified by grammar

Grammar[文法]

- Formal definition[形式化定义]: 4 components **{T, N, s, δ}**
- ***T***: set of terminal symbols[终结符]
  - Basic symbols from which strings are formed
  - Essentially **tokens** from lexer - leaves in the parse tree
- ***N***: set of non-terminal symbols[非终结符]
  - Each represents a set of strings of terminals – internal nodes
  - E.g.: declaration, statement, loop, ...
- **s**: start symbol[开始符号]
  - One of the non-terminals
- ***σ***: set of productions[产生式]
  - Specify the manner in which the terminals and non-terminals can be combined to form strings
  - "*LHS* → *RHS*": left-hand-side produces right-hand-side

Grammar (cont.)

- Usually, we can just write the σ[简写]

- Merge rules sharing the same LHS[规则合并]
  - $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \ldots, \alpha \rightarrow \beta_n$
  - $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

{T, N, s, δ}

G = ({id, +, *, (,)} , {E}, E, P)
P = { E → E + E,
E → E * E,
E → (E),
E → id }

→

G: E → E + E,
E → E * E,
E → (E),
E → id

→

E → E + E | E * E | (E) | id

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

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
  - {{id , int float ;}, {*declaration type idlist*}, *declaration*, δ}

    T — N — s
- Production rules (δ)

  declaration → *type idlist* ;

  idlist → id | *idlist* , id

  type → int | float

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

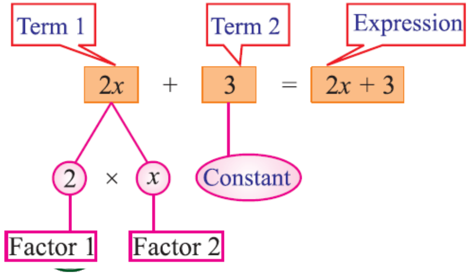

https://www.cse.iitb.ac.in/~uday/courses/cs324-07/syntax-analysis.pdf

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

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

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

Sentence

Sentential form