LINGUAGENS FORMAIS E AUTÔMATOS

Definição Informal para Linguagens

- Um conjunto, possivelmente infinito, de strings
- Vários formalismos para definir conjuntos
  - Escolha depende da importância de simplicidade vs. expressividade

Gramática e Autômato

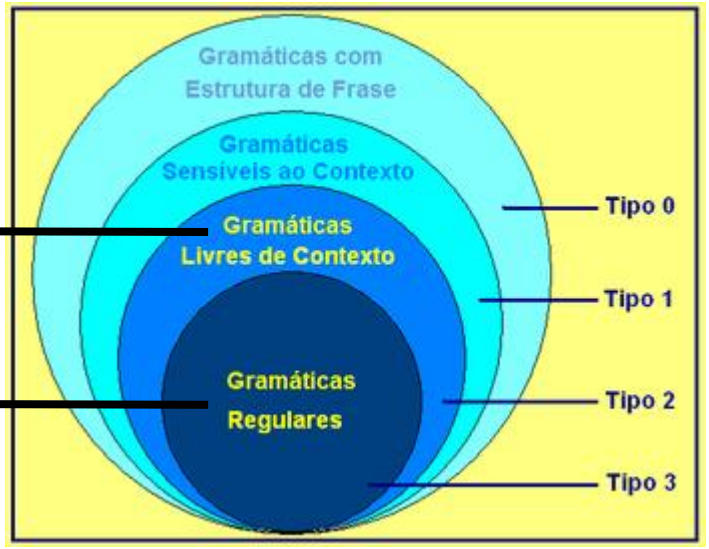
- Gramática é a expressão de uma linguagem
  - Dita regras de formação para as strings
- Autômato é uma máquina abstrata capaz de reconhecer as strings de uma linguagem
  - Um autômato diz se uma string pertence ou não a uma determinada linguagem

IF688

- Dois formalismos são de interesse em IF688
  - Linguagens Regulares
  - Linguagens Livre de Contexto

Pushdown Automata

Nondeterministic Finite Automata



Hierarquia de Chomsky

Expressão Regular

- Exemplo: números e identificadores de uma linguagem de programação

Identificadores em Pascal: **letter (letter | digit) ***

Gramática Livre de Contexto

- Um conjunto de símbolos **terminais**
- Um conjunto de símbolos **não-terminais**
- Um não terminal designado **inicial**
- Um conjunto de **produções**
  - cada produção consiste de um não-terminal, uma "seta", e uma seqüência de símbolos terminais e não terminais

Exemplo modificado

list → *list* **+** *digit*
list → *list* **–** *digit*
list → *digit*
digit → **0**
digit → **1**
…
digit → **9**

list → *list* **+** *digit* | *list* **-** *digit* | *digit*
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Deriva-se strings de uma gramática G a partir do seu símbolo inicial e repetidamente substituindo não-terminais pelo corpo de uma produção

A linguagem (conjunto de strings) reconhecida por G chama-se L(G). Inclui todas as strings que é possível se obter através de derivações em G.

Deriva-se strings de uma gramática G a partir do seu símbolo inicial e repetidamente substituindo não-terminais pelo corpo de uma produção

A linguagem (conjunto de strings) reconhecida por G chama-se L(G). Inclui todas as strings que é possível se obter através de derivações em G.

Exemplo

- Para a gramática G abaixo:

list → *list* **+** *digit* | *list* **-** *digit* | *digit*

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

L(G) = {0,1,...,0+1, 0+2, ...}

Exercício 1

- Proponha definição alternativa para evitar o uso do símbolo ε (epsilon)

call → *id* **(** *optparams* **)**
optparams → *params* | ε
params → *params* **,** *id* | *id*

Resposta

- Proponha definição alternativa para evitar o uso do símbolo ε

call → *id* **()** | *id* **(***params***)**
params → *params* **,** *id* | *id*

Exercício 2

- Defina uma gramática para a linguagem de parênteses. E.g., (), ()(), (()()),)(()), etc.

Resposta

- Defina uma gramática para a linguagem de parênteses. E.g., (), ()(), (()()), ()(()), etc.

$$A \rightarrow \mathbf{(}\ A\ \mathbf{)} \mid A\ A \mid ()$$

$$A \rightarrow \mathbf{(}\ A\ \mathbf{)} \mid A\ A \mid \varepsilon$$

A linguagem desta gramática inclui a string vazia.

LINGUAGENS DE PROGRAMAÇÃO

Especificação de uma Linguagem de Programação

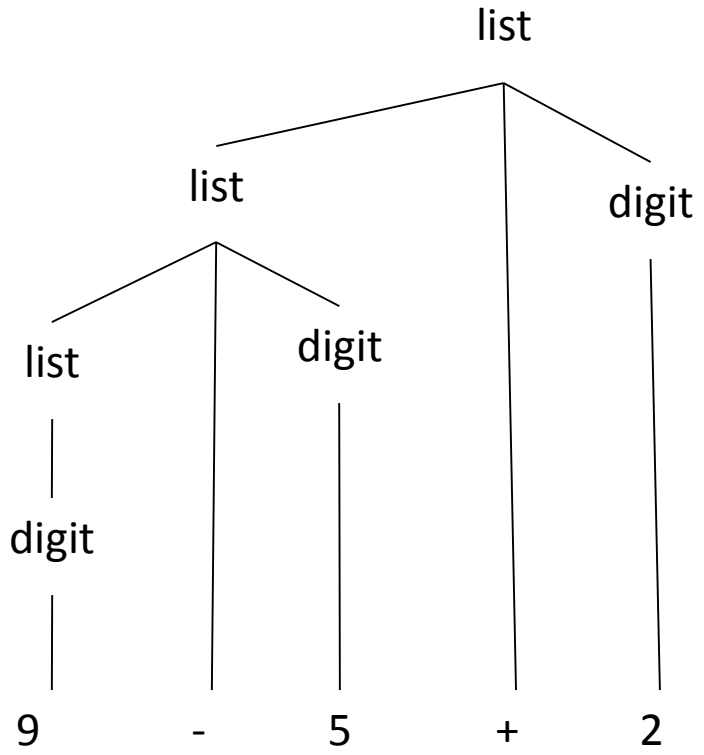
- Descrição da sintaxe
  - Tipicamente formal
- Descrição da semântica
  - Tipicamente informal
    - Mas existem formalismos: semântica operacional, denotacional, de ações, etc.

Parse Trees (Árvore Sintática)

- Árvore que descreve o processo de derivação de uma string de entrada
- Definição
  - A raiz é o símbolo inicial
  - Cada folha é um terminal ou ε
  - Cada nó interior é um não-terminal
  - Se A é um não-terminal e $X_1, X_2,\ldots,X_n$ são labels de filhos deste nó, deve haver uma produção $A \rightarrow X_1 X_2 \ldots X_n$

Exemplo

Árvore sintática para 9 – 5 + 2

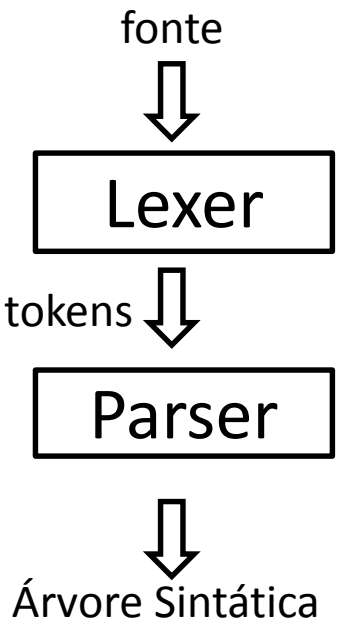


list → *list* **+** *digit* | *list* **-** *digit* | *digit*

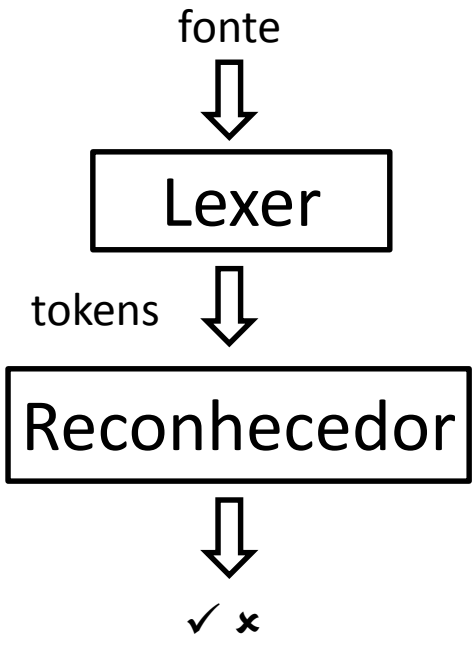
digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

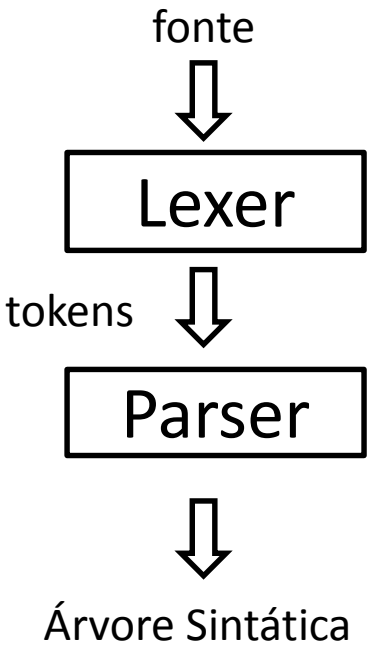
Parsing

- Parsing
  - Entrada: programa fonte
  - Saída: árvore sintática do programa fonte
  - Busca de derivação de uma string



Reconhecedores e Parsers





AMBIGUIDADE, PRECEDÊNCIA, E ASSOCIATIVIDADE

Ambiguidade

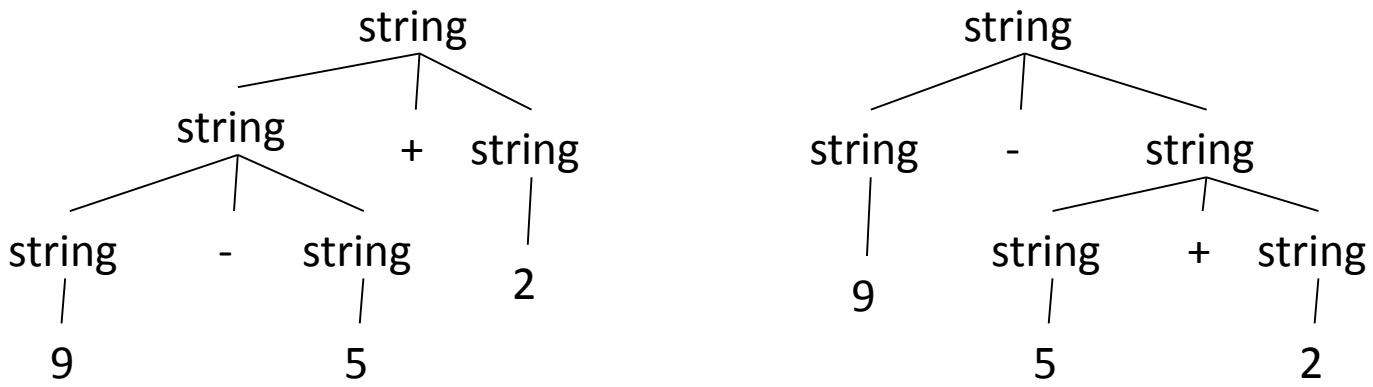
- Uma gramática **ambígua** pode gerar mais de uma parse tree para a mesma string

A interpretação pode ser diferente de acordo com a estrutura derivada!

Exemplo

string → *string* **+** *string*
| *string* **-** *string*
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Duas parse trees para a entrada “9 – 5 + 2”:



Como Eliminar Ambiguidade

- Reescrever gramática (mais comum)
- Usar gramáticas ambíguas com informações adicionais sobre como resolver ambigüidades

Precedência de operadores

- Multiplicação tem precedência sobre adição
  - Exemplo: 9 + 5 * 2 equivale a 9 + (5 * 2)

Exemplo

expr → *expr* **+** *term* | *expr* **–** *term* | *term*

term → *term* ***** *factor* | *term* **/** *factor* | *factor*

factor → *digit* | **(** *expr* **)**

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Não-terminais adicionais são usados para definir os níveis de precedência

Associatividade de Operadores

- Na maioria das linguagens de programação **+**, **–**, ***** e **/** associam à esquerda
  - Exemplo: 9 – 5 + 2 equivale a (9-5)+2
- Atribuição em C e exponenciação associam à direita
  - Exemplo: a = b = c equivale a a = (b = c)

Exemplo: associatividade à direita

right → *letter* **=** *right* | *letter*

letter → **a** | **b** | … | **z**

Exercício 3

- Modifique a gramática abaixo para que expressões aritméticas associem a esquerda

string → *string* **+** *string*
| *string* **-** *string*
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Resposta

- Modifique a gramática abaixo para que expressões aritméticas associem a esquerda

string → *string* **+** *string*
| *string* **-** *string*
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

string → *string* **+** *val* | *string* **–** *val* | *val*
val → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

PARSERS TOP-DOWN E BOTTOM-UP, BACKTRACKING

Top-down ou bottom-up parsers

- Nomes referem-se à ordem em que os nós das *parse trees* são criados
  - Top-down
    - mais fáceis de escrever à mão
  - Bottom-up
    - suportam uma classe maior de gramáticas e de esquemas de tradução;
    - mais usados por geradores de parsers

Backtracking

- Backtracking (contexto: compiladores/parsers)
  - Processo de tentativa e erro onde o parser percebe que tomou uma decisão errada

O parsing sem backtracking é chamado de **preditivo**

Exemplo

- Backtracking é necessário ao se perceber que não é possível fazer parsing de 9 + 5 * 2 a partir de *expr* → *term*

expr → *expr* **+** *term* | *expr* **–** *term* | *term*

term → *term* ***** *factor* | *term* **/** *factor* | *factor*

factor → *digit* | **(** *expr* **)**

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

PARSER PREDITIVO RECURSIVO DESCENDENTE

Parser Recursivo Descendente

- Características
  - Parser top-down e preditivo
    - Simples de se implementar (intuitivo)
    - Não funcionam para qualquer gramática!

Parser Recursivo Descendente

- Abordagem
  - Um procedimento para cada símbolo não-terminal
  - Recursão reflete recursão da gramática
  - Símbolos epsilon são tratados no contexto de uso

Exemplo

expr → *expr* **+** *term* | *expr* **–** *term* | *term*

term → *term* ***** *factor* | *term* **/** *factor* | *factor*

factor → *digit* | **(** *expr* **)**

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

stmt → **if (***expr* **)** *stmt*

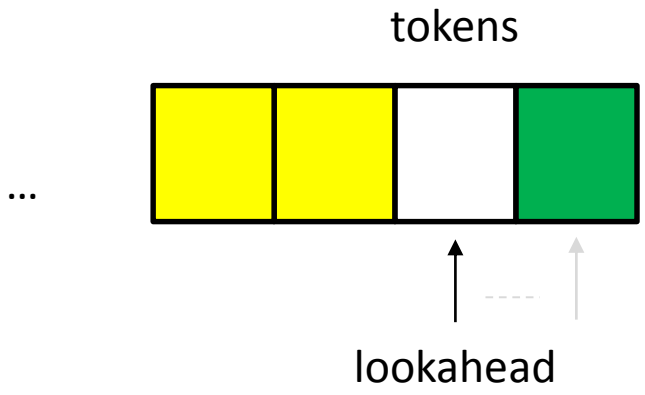
| **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*

| **other**

optexpr → *expr* | ε

Função Auxiliar

```
void match (terminal t) {
   if (lookahead == t) moveLookahead();
   else syntax_error();
 }
```



Exemplo: Recursive descent parsing

```
void stmt() {
    switch (lookahead) {
      case TO_IF:
       match(TO_IF); match(TO_OP); expr(); match(TO_CL);
       stmt(); break;
    case TO_FOR:
       match(TO_FOR); match(TO_OP); optexpr();
       match(TO_SEMI_COL);  optexpr();
       match(TO_SEMI_COL); optexpr();
       match(TO_CL); stmt(); break;
      case TO_OTHER: match(TO_OTHER); break;
      default: syntax_error();
   }
}
```

stmt → **if (** *expr* **)** *stmt*

| **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*

| **other**

Exercício 4

- Modifique a função stmt() para tratar produção epsilon abaixo

$$optexpr \rightarrow expr \mid \varepsilon$$

Resposta

```
void stmt() {
    switch (lookahead) {
      case TO_IF:
       match(TO_IF); match(TO_OP); expr(); match(TO_CL);
       stmt(); break;
    case TO_FOR:
       match(TO_FOR); match(TO_OP);  if (lookahead != TO_SEMI_COL)
    expr();
       match(TO_SEMI_COL); if (lookahead != TO_SEMI_COL) expr();
       match(TO_SEMI_COL); if (lookahead != TO_CL) expr();
       match(TO_CL); stmt(); break;
      case TO_OTHER: match(TO_OTHER); break;
      default: syntax_error();
   }
}
```

stmt → **if (***expr* **)** *stmt*

| **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*

| **other**

Exercício 5

- Implemente a função recursiva para *factor*

$$factor \rightarrow digit \mid \mathbf{(}\ expr\ \mathbf{)}$$

Resposta

- Implemente a função recursiva para *factor*

```
void factor( ) {
    switch (lookahead) {
      case TO_OP:
         match(TO_OP); expr(); match(TO_CL);  break;
      case TO_DIGIT: match(TO_DIGIT); break;
      default: syntax_error();
   }
}
```

Problema do parser recursivo descendente

- Recursão à esquerda leva a loop infinito:

  expr → expr + term | term

O parser permanece aplicando a mesma produção sem consumir token algum!

Solução

- Elimine recursão a esquerda com reescrita
- Exemplo (“ba...a”)
  - Reescreva $A \rightarrow Aa \mid b$ como:

$$A \rightarrow bR$$
$$R \rightarrow aR \mid \varepsilon$$

Na prática, porém, trabalhoso devido a regras de precedência e associatividade.

Exercício 6

- Elimine recursões a esquerda

expr → *expr* **+** *term* | *expr* **–** *term* | *term*

term → *term* ***** *factor* | *term* **/** *factor* | *factor*

factor → *digit* | **(** *expr* **)**

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Tentativa...

- Elimine recursões a esquerda

expr → *term* **+** *expr* | *term* **–** *expr*| *term*

term → *factor* ***** *term*| *factor* **/** *term*| *factor*

factor → *digit* | **(** *expr* **)**

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

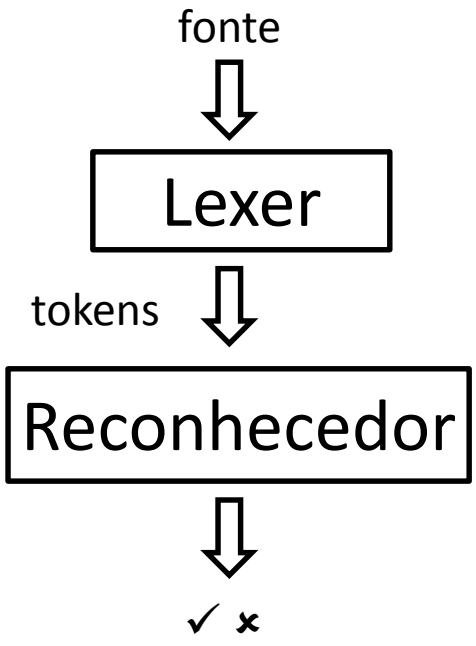
Incorreto!

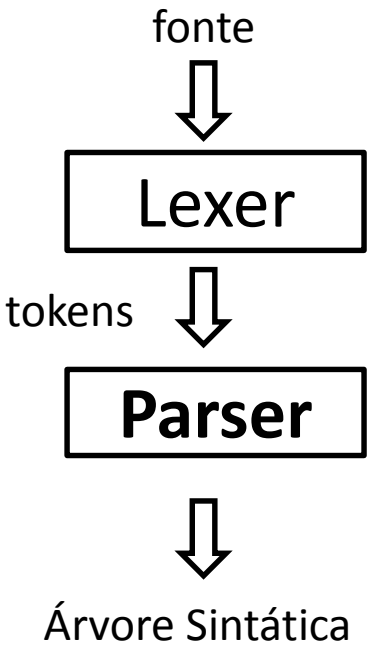
Modifica regra de associatividade.

Modificação manual é muitas vezes trabalhosa!
Outros tipos de parsers permitem definir regras de precedência em produções.

CONSTRUÇÃO DA ÁRVORE E NAVEGAÇÃO

Ainda não construímos um parser

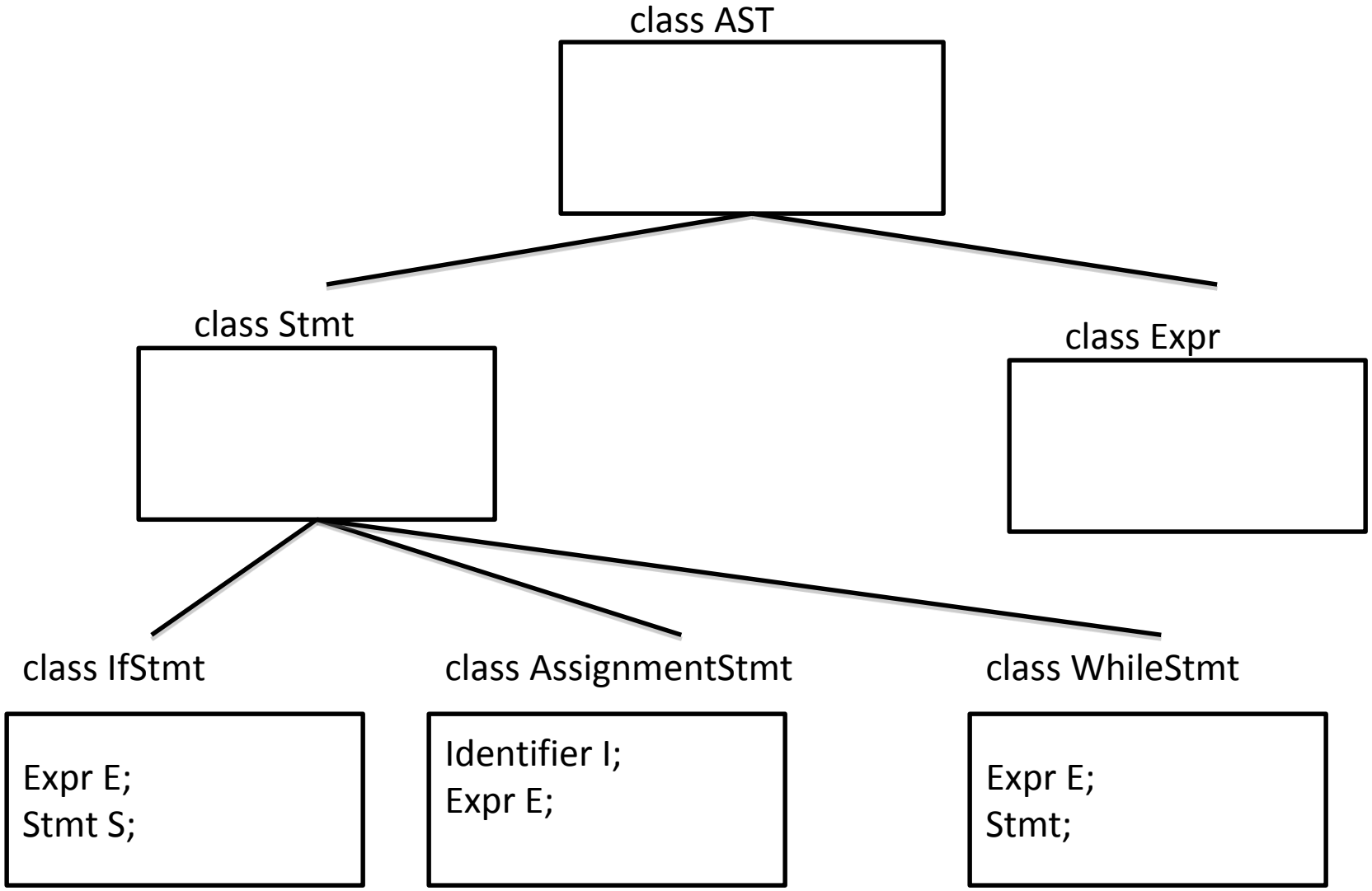




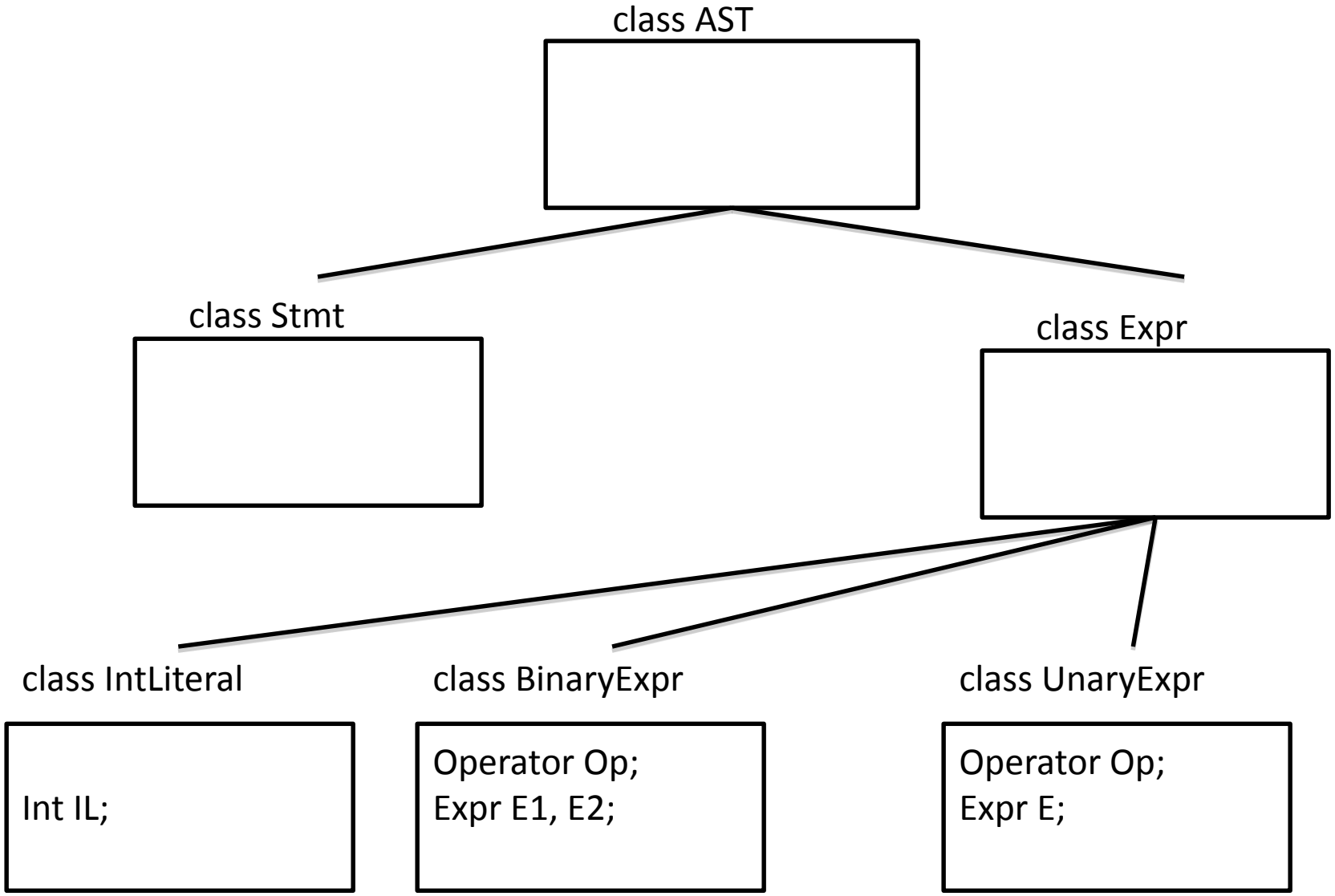
Construção da Árvore Sintática

- É preciso definir tipos para nós da árvore
  - E.g., classes para *expr* e *stmt* no exemplo anterior
  - Parsing gera objetos destes tipos!
- Árvore **abstrata** vs. **concreta**
  - Abstrata ignora distinções superficiais ou implícitas
    - E.g., parênteses, ponto-e-vírgula, e espaços em branco

Exemplo: hierarquia de classes



Exemplo: hierarquia de classes



Exercício 7

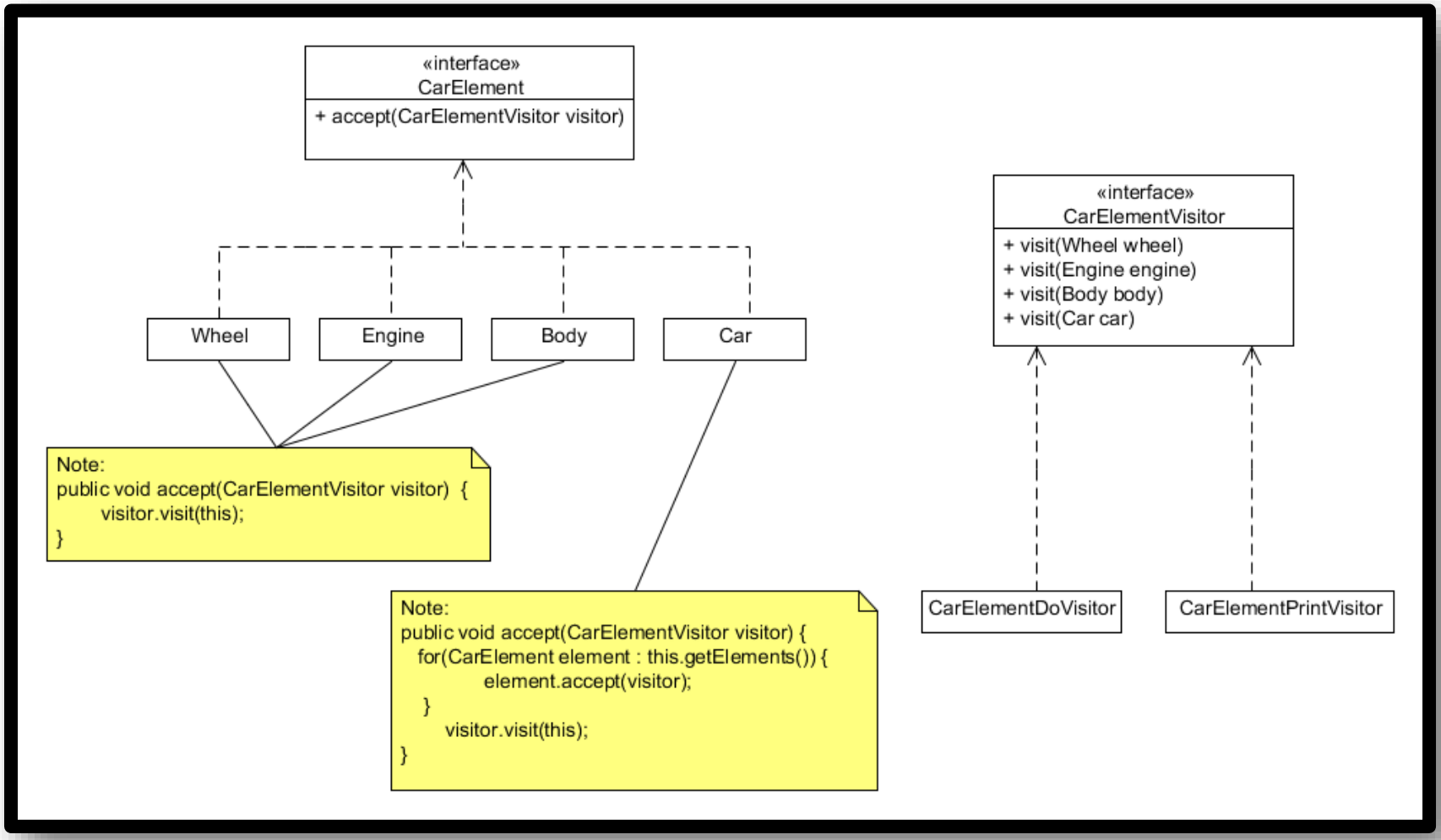
- Elabore (manualmente) a árvore sintática e a árvore concreta para a expressão (5 + 3) * 2

Exploração dos nós da árvore

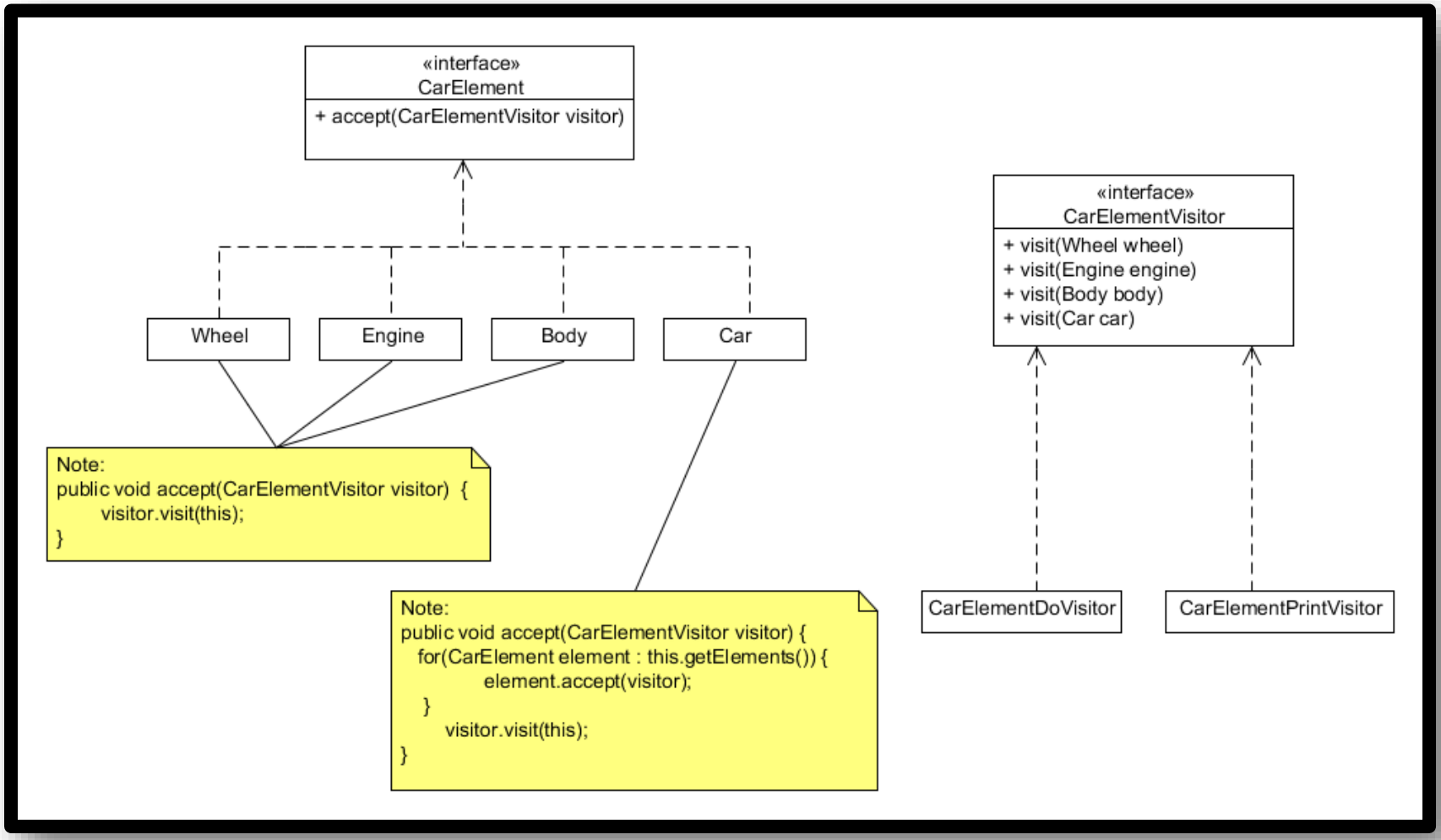
- Padrão de projeto **Visitor*** define como visitar nós de uma estrutura hierárquica
  - Tipo de retorno deve ser consistente
  - Bastante comum em compiladores

* Design Patterns: Elements of Reusable Object-Oriented Software. Gamma e outros.

Visitor design pattern



Visitor design pattern



Exercício 8

- Imprima expressões aritméticas em notação pós-fixada a partir de suas árvores sintáticas
  - Use definições abaixo
  - Explore a árvore em uma determinada ordem

```
interface Expr {…}
class BinaryExpr implements Expr {
  Operator op;  Expr exp1, exp2;…
}
class Digit implements Expr {  … }
class Operator {…}
```

Resposta

```
interface Expr {
  void accept(Visitor vis); }
class BinaryExpr implements Expr {
  Operator op;  Expr exp1, exp2;
  void accept(Visitor vis) {
    exp1.accept(vis);
    exp2.accept(vis);
    op.accept(vis);
  } }
class Digit implements Expr {  int val;
   void accept(Visitor vis) {  vis.visit(this);  }
 }
class Operator {  char val;
 void accept(Visitor vis) { vis.visit(this); }
}
```

```
interface Visitor {
  void visit(Expr p);
  void visit(BinaryExpr p);
  void visit(Digit p);
  void visit(Operator p);
}
```

```
class PosFixPrinter implements Visitor {
  StringBuffer sb = new StringBuffer();
  void visit(Expr p) {}
  void visit(BinaryExpr p) {}
  void visit(Digit p){ sb.append(p.val); }
  void visit(Operator p) { sb.append(p.val); }
}
```

Exercício 9

- A resposta anterior definiu a ordem de busca diretamente no método Binary.accept. Isto é incomum, pois o método accept normalmente não deve ser modificado. Reescreva o visitor anterior assumindo a seguinte definição para o método accept.

```
void accept(Visitor vis) {
   exp1.accept(vis);
   op.accept(vis);
   exp2.accept(vis);
}
```