Análise Sintática (parte 2)

FONTES

Sumário da Aula

- Veremos alguns conceitos adicionais ligados à fase de análise sintática
  - **Ambiguidade**
  - **Tratando Ambiguidades**
  - **Sintaxe abstrata x Sintaxe concreta**

Ambiguidade

FONTES

Ambiguidade

- Uma cadeia (sequencia) pode ser **gerada ambiguamente** por uma gramática sse:
  - Existem duas árvores de derivação que geram (exatamente) a mesma cadeia
- Dizemos que uma **gramática é ambígua** sse
  - Existe alguma cadeia que é gerada ambiguamente por esta gramática

Gramáticas Ambíguas

- Em outras palavras, uma **gramática é ambígua** se ela pode gerar uma mesma cadeia com duas árvores de derivação distintas
  - Não precisa ter duas árvores para <u>todas</u> as cadeias
  - Basta acontecer com <u>uma</u> cadeia

Exemplo

- Na gramática abaixo, mostrar duas árvores para a cadeia "`1+x*10`"

```
expressão ::= expressão + expressão
            | expressão - expressão
            | expressão * expressão
            | expressão / expressão
            | ( expressão )
            | IDENTIFICADOR
            | INTEIRO-LITERAL
```

```
E -> E+E | E-E | E*E | E/E
E -> (E) | IDENTIFICADOR
         | INTEIRO-LITERAL
```

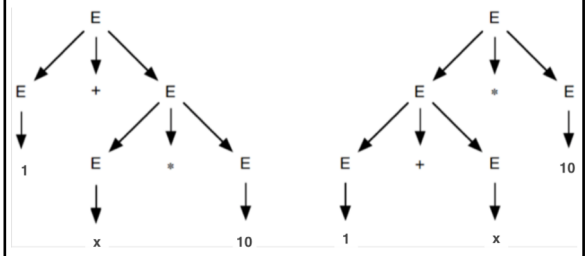
Exemplo

- Na gramática abaixo, mostrar duas árvores para a cadeia "1+x*10"

```
E -> E + E
  -> INT. + E
  -> INT. + E * E
  -> INT. + ID. * E
  -> INT. + ID * INT.
```

```
E -> E * E
  -> E + E * E
  -> INT. + E * E
  -> INT. + ID. * E
  -> INT. + ID * INT.
```

Exemplo



Sintaxe Abstrata

- Quais as **árvores de derivação** possíveis para estas entradas?

"**1 + 2 * 3**"

"**3 + 1 - 2**"

"**3 * 2 / 4**"

Gramáticas Ambíguas

- Gramáticas ambíguas são, em geral, inadequadas para uso em compiladores
  - Dificultam a construção do analisador sintático
  - Induzem mais de uma semântica para o código
    - compiladores distintos podem gerar códigos finais que funcionam diferentemente
- É necessário fornecer informações adicionais (além da gramática)

Informações Adicionais

- Dois tipos de informação costumam ser dadas junto com a gramática
  - **Precedência de operadores**
  - **Associatividade de operadores**

Precedência

- Precedência diz qual operação binária será aplicada primeiro
  - Exemplo: como interpretar **a + b * c + d * e** ?
  - Se o operador * tiver maior precedência:
    ((a + (b * c)) + (d * e))
  - Se o operador + tiver maior precedência:
    (((a + b) * (c + d)) * e)

Associatividade

- Associatividade diz a ordem em que serão realizadas operações de mesma precedência
  - Exemplo: como interpretar **a + b + c + d + e** ?
  - Se o operador for associativo à esquerda:
    ((((a + b) + c) + d) + e)
  - Se o operador for associativo à direita:
    (a + (b + (c + (d + e))))

Tratando Ambiguidades

FONTES

Tratando Ambiguidades

- Veremos como tratar dois tipos de ambiguidades:
  - Ambiguidade do “else”
  - Ambiguidade de expressões

Tratando Ambiguidades

- A seguinte definição de comando apresenta ambiguidade

```
cmd ::= if ( expr ) cmd else cmd
      | if ( expr ) cmd
      | nop

expr ::= x
```

- Exemplo: “if (x) if (x) nop else nop”
  - A qual “if” está ligado o “else”?

Tratando Ambigüidades

- O mais comum é considerar que o “else” casa com o “if” aberto mais próximo
- Geralmente, a gramática anterior funciona bem com as técnicas de parser que vamos aprender
  - Na técnica LR, deve surgir um ou outro conflito, facilmente resolvida
- Mas é possível remover essa ambigüidade...

Tratando Ambigüidades

- A estratégia é classificar os comandos em comandos **incompletos** (ou abertos) e **completos**
- Comandos completos (*matched*)
  - if-else
  - todos os outros
- Comandos incompletos ou abertos (*open*)
  - if (sem else)
  - if-else com um comando incompleto após o else

Tratando Ambigüidades

- Solução

```
cmd ::= matched-cmd
      | open-cmd

matched-cmd ::=
      | if ( expr ) matched-cmd else matched-cmd
      | outro

open-cmd ::=
      | if ( expr ) cmd
      | if ( expr ) matched-cmd else open-cmd
```

Tratando Ambigüidades

- Como mostrado, a gramática abaixo é ambígua, mas essa ambigüidade pode ser resolvida com regras de precedência e associatividade

```
expressão ::= expressão + expressão
            | expressão - expressão
            | expressão * expressão
            | expressão / expressão
            | expressão ^ expressão
            | ( expressão )
            | INTEIRO-LITERAL
```

- Como incluir essas regras na própria gramática?

Tratando Ambigüidades

- Criar um não-terminal para cada nível de precedência, da menor para a maior
  - Chamaremos de: **exprA**, **exprB**, **exprC**...
- No último nível ficam apenas expressões não binárias (como se tivessem máxima precedência)

Tratando Ambigüidades

- O não-terminal de cada nível tem uma produção para cada operador do nível
  - Exemplo: **exprA** terá uma produção para + e outra para -
- Para cada operador, os operandos serão:
  - o não-terminal do nível atual
  - e o não-terminal do próximo nível (nível de maior precedência)
  - Mas em que ordem?

Tratando Ambigüidades

- Ordem dos operandos em cada produção
  - Se for **associativo à esquerda**, colocar o nível atual como operando da esquerda e o próximo nível, no lado direito
  - Se for **associativo à direita**, colocar o nível atual como operando da direita (e o próximo nível, na esquerda)
- Para reconhecer o caso de não haver nenhuma operação do nível, deve haver uma produção de direto para o próximo nível

Tratando Ambigüidades

- No exemplo anterior, vamos assumir três níveis para os operadores binários
  - As demais expressões formam um quarto nível
- Ordenados da menor para a maior precedência:
  - adição e subtração, associativos à esquerda
    - **exprA**
  - multiplicação e divisão, assoc. à esquerda
    - **exprB**
  - exponenciação, assoc. à direita
    - **exprC**

Tratando Ambigüidades

- Resultado

```
expressão ::= exprA

exprA ::= exprA + exprB
        | exprA - exprB
        | exprB

 ...
```

associativos à esquerda

Tratando Ambigüidades

- Resultado (continuação)

```
exprB ::= exprB * exprC
        | exprB / exprC
        | exprC

exprC ::= exprD ^ exprC
        | exprD

exprD ::= ( expressão )
        | INTEIRO-LITERAL
```

associativos à esquerda

associativo à direita

Sintaxe Abstrata x Concreta

FONTES

Sintaxe Abstrata

- É comum uma linguagem ser especificada por meio de uma gramática de **sintaxe abstrata**
  - Em geral, ela é ambígua
  - Porém, é mais simples de entender
- A especificação da linguagem costuma descrever, à parte, as precedências e associatividades dos operadores

Exemplo

- Considere como uma linguagem de programação:

```
expressão ::= expressão + expressão
            | expressão * expressão
            | ( expressão )
            | IDENTIFICADOR
            | INTEIRO-LITERAL
```

  - Já vimos como ela é ambígua e porque isso prejudica a criação do compilador

Sintaxe Concreta

- Por outro lado, a gramática tal como ela foi usada para implementar o parser é chamada de **sintaxe concreta**
  - Geralmente não tem ambigüidades
  - Mais fácil de implementar
- Específica da implementação
  - Cada compilador (de uma mesma linguagem) pode criar uma diferente

Sintaxe Concreta

- Exemplo de sintaxe concreta correspondente à gramática de expressões anterior

```
expressão ::= termo + expressão
            | termo

termo ::= fator * termo
        | fator

fator ::= ( expressão )
        | INTEIRO-LITERAL
        | IDENTIFICADOR
```

Tratando Ambiguidades

- No fundo, na maior parte desta aula estávamos vendo a partir de **sintaxe abstrata** para prepará-la para ser usada como **sintaxe concreta** (em um parser)