Compiladores – Análise Semântica

Fabio Mascarenhas – 2015.2

http://www.dcc.ufrj.br/~fabiom/comp

Árvores Sintáticas Abstratas (ASTs)

- A árvore de análise sintática tem muita informação redundante
  - Separadores, terminadores, não-terminais auxiliares (introduzidos para contornar limitações das técnicas de análise sintática)
- Ela também trata todos os nós de forma homogênea, dificultando processamento deles
- A árvore sintática abstrata joga fora a informação redundante, e classifica os nós de acordo com o papel que eles têm na estrutura sintática da linguagem
- Fornecem ao compilador uma representação compacta e fácil de trabalhar da estrutura dos programas

Exemplo

- Seja a gramática abaixo:

```
E -> n
   | ( E )
   | E + E
```

- E a entrada `25 + (42 + 10)`

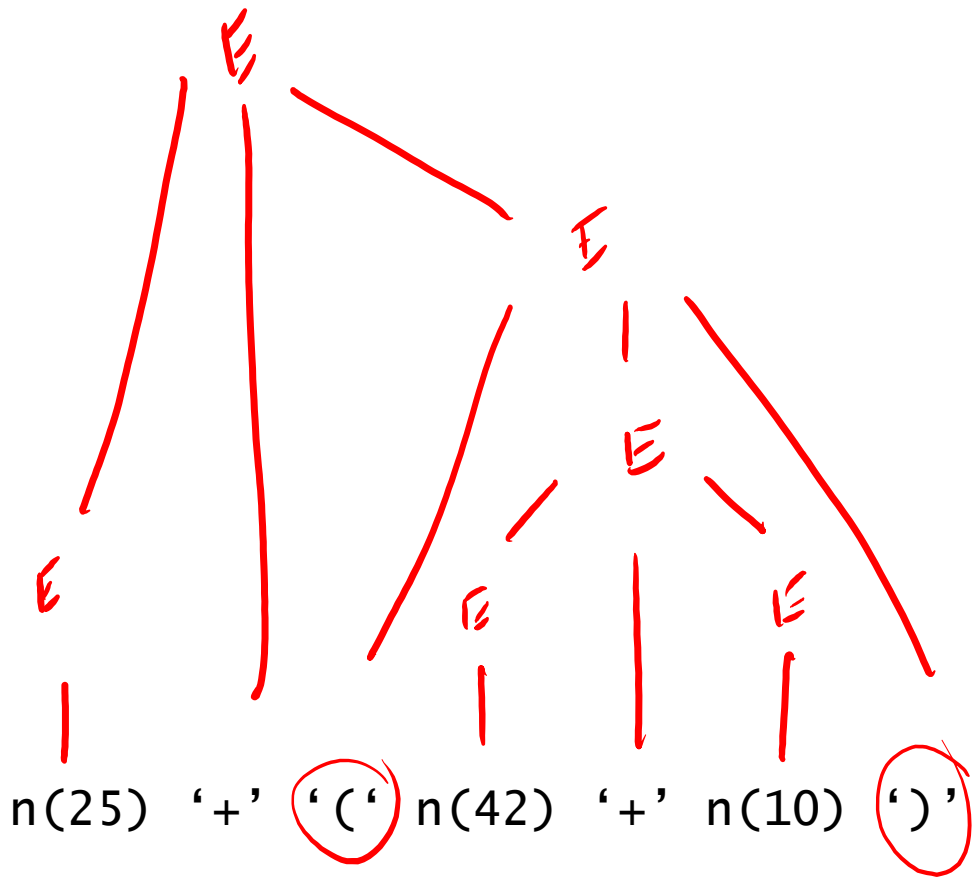
- Após a análise léxica, temos a sequência de tokens (com os lexemes entre parênteses):

`n(25)` `'+'` `'('` `n(42)` `'+'` `n(10)` `')'` EOF

- Um analisador sintático bottom-up construiria a árvore sintática da próxima página

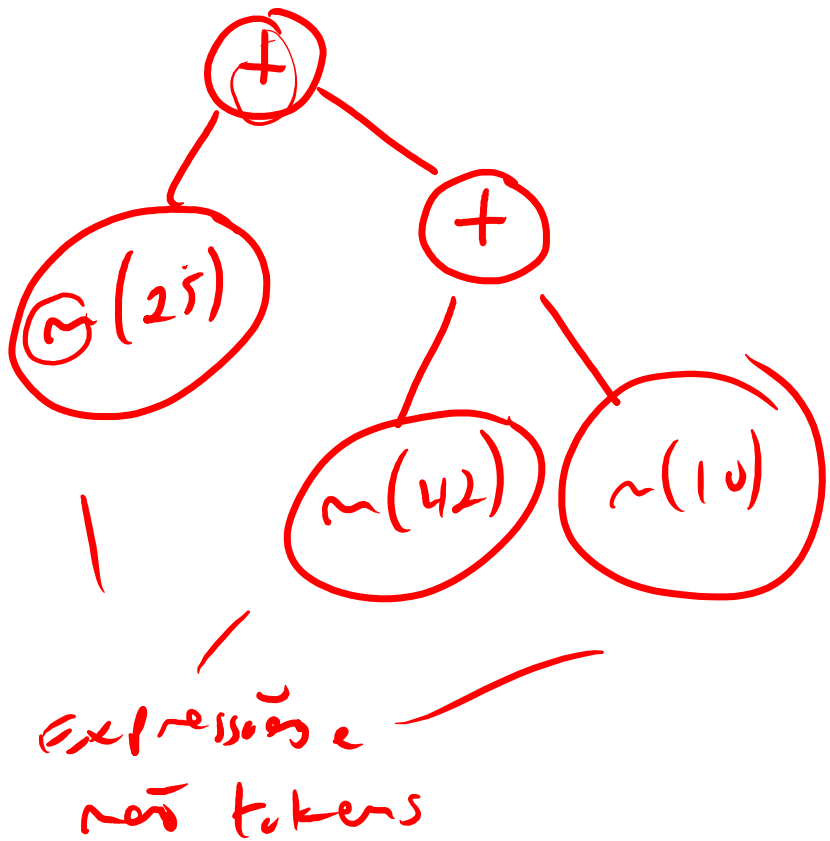
Exemplo – árvore sintática

```
E -> n
   | ( E )
   | E + E
```



Exemplo - AST

```
E -> n
   | ( E )
   | E + E
```



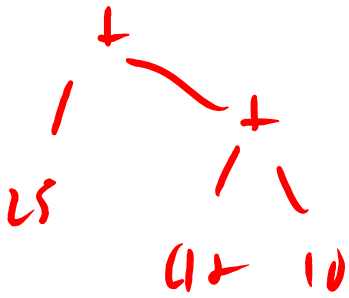

```
n(25) '+' '(' n(42) '+' n(10) ')'
```

Representando ASTs

- Cada estrutura sintática da linguagem, normalmente dada pelas produções de sua gramática, dá um tipo de nó da AST
- Em um compilador escrito em Java, vamos usar uma classe para cada tipo de nó
- Não-terminais com várias produções ganham uma interface ou uma classe abstrata, derivada pelas classes de suas produções
- Nem toda produção ganha sua própria classe, algumas podem ser redundantes

```
E -> n          => Num (deriva de Exp)
   | ( E )      => Redundante
   | E + E      => Soma (deriva de Exp)
```

Exemplo – Representando a AST



```
interface Exp {}

class Num implements Exp {
  int val;
  Num(String lexeme) {
    val = Integer.parseInt(lexeme);
  }
}

class Soma implements Exp {
  Exp e1;
  Exp e2;

  Soma(Exp _e1, Exp _e2) {
    e1 = _e1; e2 = _e2;
  }
}
```

Uma AST para TINY

- Vamos lembrar da gramática SLR de TINY:

```
TINY -> CMDS
CMDS -> CMDS ; CMD
      | CMD
CMD  -> if EXP then CMDS end
      | if EXP then CMDS else CMDS end
      | repeat CMDS until EXP
      | id := EXP
      | read id
      | write EXP
```

```
EXP  -> EXP < EXP
      | EXP = EXP
      | EXP + EXP
      | EXP - EXP
      | EXP * EXP
      | EXP / EXP
      | ( EXP )
      | num
      | id
```

- Vamos representar listas (CMDS) usando a própria interface `List<T>` de Java

Uma AST para TINY - Resumo

- Duas interfaces: `Cmd`, `Exp`
- As duas produções do `if` compartilham o mesmo tipo de nó da AST
- Quatorze classes concretas
- Poderíamos juntar todas as operações binárias em uma única classe, e fazer a operação ser mais um campo
- Ou poderíamos ter separado `If` e `IfElse`
- Não existe uma maneira certa; a estrutura da AST é engenharia de software, não matemática

MiniJava

- Vocês estão implementando um compilador MiniJava como trabalho dessa disciplina

- MiniJava possui classes com herança simples, e métodos que podem ser redefinidos nas subclasses; um programa é um conjunto de classes

- O fragmento de gramática abaixo dá a estrutura dos programas MiniJava

```
PROG   -> MAIN {CLASSE}
MAIN   -> class id '{' public static void main
          ( String [ ] id ) '{' CMD '}' '}'
CLASSE -> class id [extends id] '{' {VAR} {METODO} '}'
VAR    -> TIPO id ;
METODO -> public TIPO id '(' [PARAMS] ')' '{' {VAR} {CMD} return EXP ; '}'
PARAMS -> TIPO id {, TIPO id}
```

AST de MiniJava

- O número de elementos sintáticos de MiniJava é bem mais extenso que as de TINY, então a quantidade de elementos na AST também será maior

- Um `Programa` tem uma lista de `Classe`, sendo que uma delas é a principal, de onde tiramos o `corpo` do programa, com apenas um `Cmd`, e o nome do parâmetro com os argumentos de linha de comando

- Uma `Classe` tem uma lista de `Var` e uma lista de `Metodo`

- Um `Metodo` tem uma lista de `Var` e um corpo com uma lista de `Var`, uma lista de `Cmd`, e uma `Exp` de retorno

- Uma `Var` tem um tipo e um nome, que são strings; `Cmd` e `Exp` são interfaces com uma série de implementações concretas

Análise Semântica

- Muitos erros no programa não podem ser detectados sintaticamente, pois precisam de *contexto*
  - Quais variáveis estão em escopo, quais os seus tipos
- Por exemplo:
  - Todos os nomes usados foram declarados
  - Nomes não são declarados mais de uma vez
  - Tipos das operações são consistentes

Escopo

- Amarração dos *usos* de um nome com sua *declaração*
  - Onde nomes podem ser variáveis, funções, métodos, tipos...
- Passo de análise importante em diversas linguagens, mesmo linguagens “de script”
- O *escopo* de um identificador é o trecho do programa em que ele está visível
- Se os escopos não se sobrepoem, o mesmo nome pode ser usado para coisas diferentes

Declarações e escopo em TINY

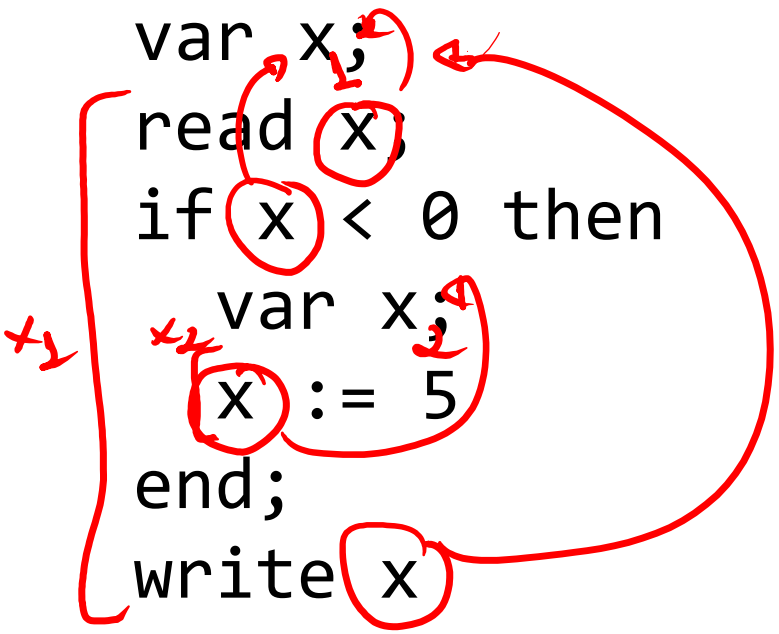
- Vamos adicionar declarações de variáveis em TINY no início de cada bloco, usando a sintaxe:

```
CMDS -> CMDS ; CMD
      | VAR CMD
VAR  -> var IDS ;
      |
IDS  -> IDS , id
      | id
```

- O escopo de uma declaração é todo o bloco em que ela aparece, incluindo outros blocos dentro dele!
- Uma variável pode ser redeclarada em um bloco dentro de outro, nesse caso ela *oculta* a variável do bloco mais externo

Exemplo - escopo

- Qual o escopo de cada declaração de *x* no programa abaixo, e qual declaração corresponde a cada uso?



```
var x;
read x;
if x < 0 then
  var x;
  x := 5
end;
write x
```