

Compiladores – Análise Semântica

Fabio Mascarenhas - 2013.2

<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:

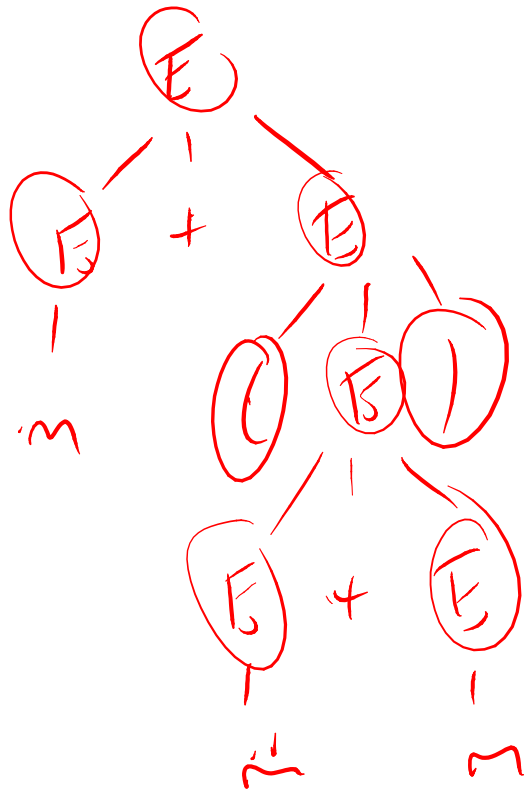
$$\begin{array}{l} E \rightarrow n \\ \quad | (F) \\ \quad | E + E \end{array}$$
ambigüidade

- E a entrada $25 + (42 + 10)$
- Após a análise léxica, temos a sequência de tokens (com os lexemes entre parênteses):

$n(25) \text{ ' + ' } ' (' \text{ } n(42) \text{ ' + ' } n(10) \text{ ') ' }$

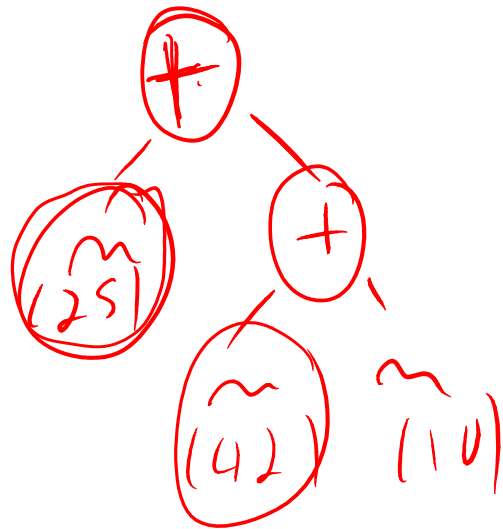
- Um analisador sintático bottom-up construiria a árvore sintática da próxima página

Exemplo – árvore sintática

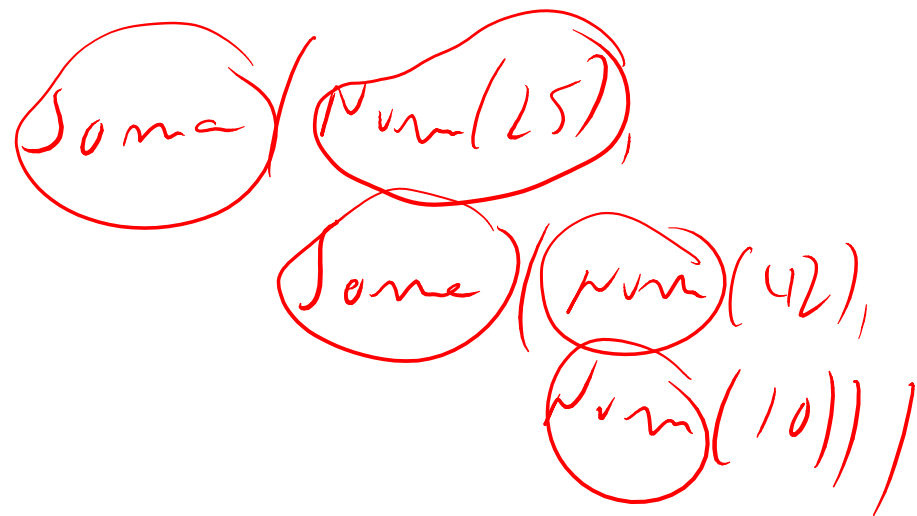


} nós

Exemplo - AST

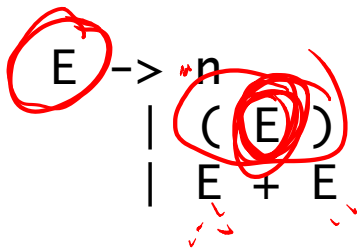


5 nís



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



=> Num (deriva de Exp)
=> Redundante
=> Soma (deriva de Exp)

~~Parêntese~~

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

{cmd}

- Vamos lembrar da gramática SLR de TINY:

<div>TINY</div> <div>CMDS</div> <div>CMD</div>	<div>-> CMDS</div> <div>-> CMDS ; CMD</div> <div> CMD</div> <div>-> if EXP then CMDS end</div> <div> if EXP then CMDS else CMDS end</div> <div> repeat CMDS until EXP</div> <div> id := EXP</div> <div> read id</div> <div> write EXP</div>	<div></div> <div></div> <div></div> <div>) if</div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div>	<div>EXP</div> <div>-> EXP < EXP</div> <div> EXP = EXP</div> <div> EXP + EXP</div> <div> EXP - EXP</div> <div> EXP * EXP</div> <div> EXP / EXP</div> <div> (EXP)</div> <div> num</div> <div> id</div>
--	--	---	--

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

List<Cmd>


números de linha!

Uma AST para TINY - Resumo

- Três interfaces: Cmd, Cond, 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 o if em If e IfElse
- Não existe uma maneira certa; a estrutura da AST é engenharia de software, não matemática

MiniJava

- MiniJava é um subconjunto de Java que vamos usar como contraponto a TINY, e como exemplo de como é a estrutura de Análise Semântica de uma linguagem OO simples
- A linguagem 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 Param e um corpo com uma lista de Var, uma lista de Cmd, e uma Exp de retorno
- Tanto uma Var quanto um Param têm um Tipo e um nome; Tipo, 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*
~~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

→ muitas vezes em tempo de execução

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 sobrepõem, 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

VAR CMD ; CMD ; CMD ; CMD

*[if ~ then
x [var x ;
i
a2
do
end*

- 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

*VAR x ;
[if ~ then
var x ;
[x
a2
end*

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;
```

imprime o valor
de x que foi lido

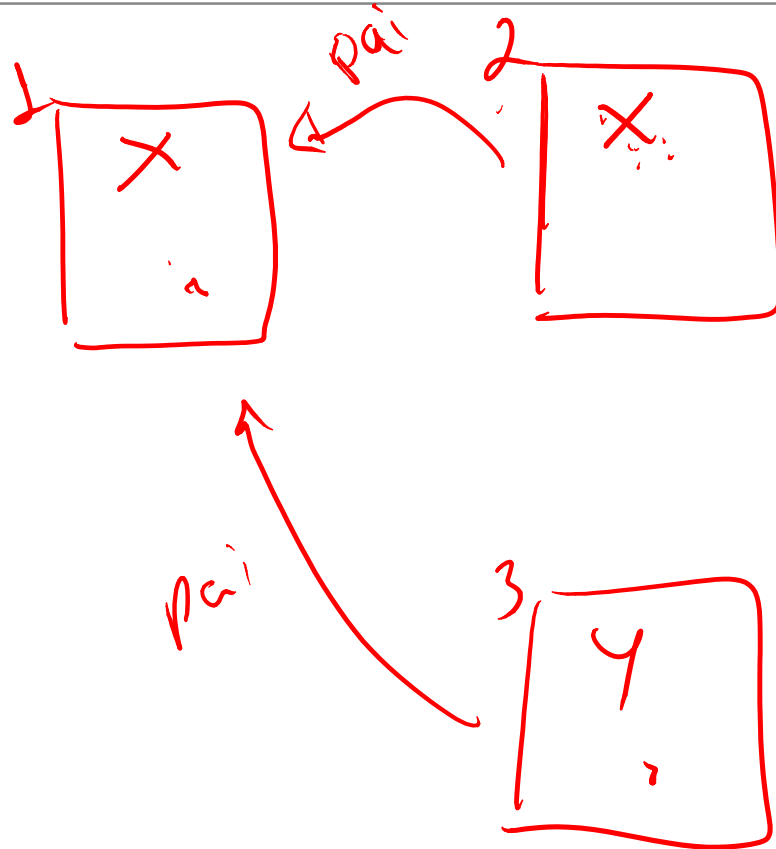
Analizando escopo

- Fazemos a análise do escopo usando uma tabela de símbolos encadeada
- Uma tabela de símbolos mapeia um *nome* a algum *atributo* desse nome (seu tipo, onde ele está armazenado em tempo de execução, etc.)
- Cada tabela corresponde a um escopo, e elas são ligadas com a tabela responsável pelo escopo onde estão inseridas
- Existem duas operações básicas: *inserir* e *procurar*, usadas na declaração e no uso de um nome
- Essas operações implementam as regras de escopo da linguagem

Tabelas de Símbolos Encadeadas

```
1 var x;  
  read x;  
  if x < 0 then  
2    var x;  
    x := 5  
  end;  
  repeat  
3    var y;  
    y := x;  
    write y;  
    x := x - 1  
  until y = 0  
  write x
```

var + \odot
erro



Procedimentos e escopo global

- Agora vamos adicionar *procedimentos* a TINY, usando a sintaxe abaixo:

```
TINY  -> PROCS ; CMDS
      | CMDS
PROCS -> PROCS ; PROC
      | PROC
PROC  -> procedure id ( ) CMDS end
CMD   -> id ( )
      | ...
```

- Nomes de procedimentos vivem em um *espaço de nomes* separado do nome de variáveis, e são visíveis em todo o programa
- Variáveis visíveis em todo o bloco principal do programa também são visíveis dentro de procedimentos (variáveis globais)

Exemplo – escopo de procedimentos

- Procedimentos podem ser mutuamente recursivos

```
procedure par()  
  if 0 < n then  
    n := n - 1;  
    impar()  
  else  
    res := 1  
  end  
end;
```

```
procedure impar()  
  if 0 < n then  
    n := n - 1;  
    par()  
  else  
    res := 0  
  end  
end;
```




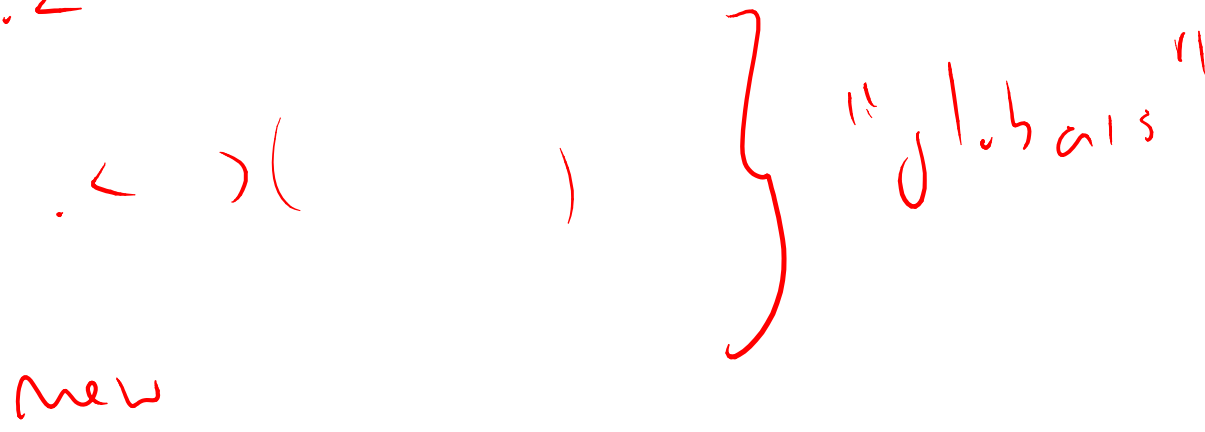
```
var x, n, res;  
read x;  
n := x;  
par();  
write res;  
n := x;  
impar();  
write res
```

Analizando escopo global

- Para termos escopo global, precisamos fazer a análise semântica em duas *passadas*
 - A primeira coleta todos os nomes que fazem parte do escopo global, e detecta declarações duplicadas
 - A segunda verifica se todos os nomes usados foram declarados
- A primeira passada constrói uma tabela de símbolos que é usada como entrada para a segunda
- No caso de TINY, essa tabela de símbolos é diferente da que usamos para variáveis

Escopos em MiniJava

- MiniJava tem vários tipos de nomes:

- Variáveis 
- Campos 
- Métodos 
- Classes 

- Cada um desses tem suas regras de escopo; alguns compartilham espaços de nomes, outros têm espaços de nomes separados

Classes

- O escopo das classes é *global*
- Uma classe é visível no corpo de qualquer outra classe
- Classes estão em seu próprio espaço de nomes

```
class Foo {  
  Bar Bar;  
}
```

```
class Bar {  
  Foo Foo;  
}
```

Variáveis e campos

- Variáveis e campos compartilham o mesmo espaço de nomes, mas as regras de escopo são diferentes
- O escopo de variáveis locais é o escopo de bloco tradicional
- O escopo de campos respeita a *hierarquia de classes* de MiniJava, uma relação dada pelas cláusulas *extends* usadas na definição das classes
- Um campo de uma classe é visível em todos os métodos daquela classe e de *todas as suas subclasses, diretas ou indiretas*
- Variáveis locais ocultam campos, mas campos não podem ser redefinidos nas subclasses

Exemplo – escopo de variáveis e campos

- O escopo do campo `x` inclui todas as subclasses de `Foo`

```
class Foo {  
    int x;  
}  
  
class Bar extends Foo { }  
  
class Baz extends Bar {  
    int m1() {  
        return x;  
    }  
  
    int m2(boolean x) {  
        return x;  
    }  
}
```

class Xyz {
 // x não é visível
}

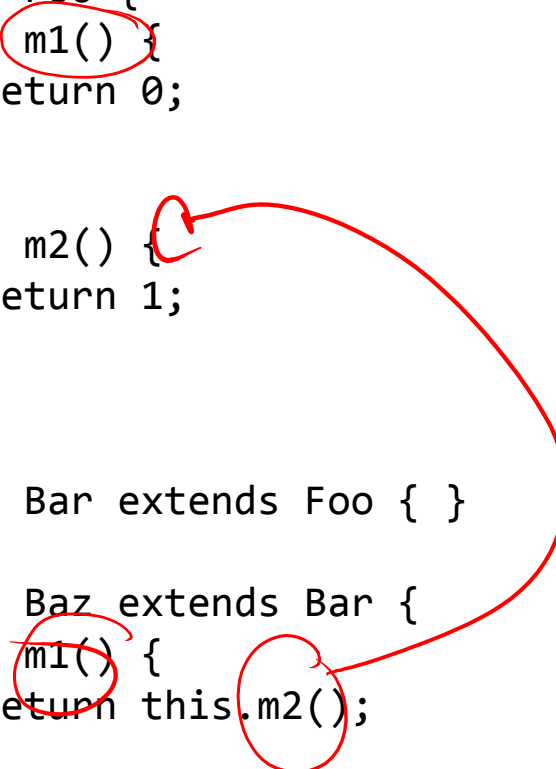
Métodos

- Como classes, métodos estão em seu próprio espaço de nomes
- Mas, como campos, o escopo de um método é a classe em que está definido e suas subclasses
- Um método não pode ser definido duas vezes em uma classe, mas pode ser redefinido em uma subclasse *contanto que a assinatura seja a mesma*
- A *assinatura* do método é o seu tipo de retorno, seu nome e os tipos dos seus parâmetros, na ordem na qual eles aparecem


Exemplo - métodos

- O método *m2* é visível em Baz, que redefine *m1*

```
class Foo {  
    int m1() {  
        return 0;  
    }  
  
    int m2() {  
        return 1;  
    }  
}  
  
class Bar extends Foo { }  
  
class Baz extends Bar {  
    int m1() {  
        return this.m2();  
    }  
}
```



Escopo em TINYPy

- O escopo em TINYPy tem algumas diferenças em relação ao de TINY
- O espaço de nomes das funções é separado das variáveis, e uma função é visível em todo o programa
- As variáveis do bloco principal da função *main* também são visíveis em todo o programa, como as globais de TINY 
- Todos os nomes são *case-insensitive*: a forma mais simples é converter para uma forma canônica (maiúsculas ou minúsculas) na inserção e consulta às tabelas de símbolos

Foo é foo é fOo é fOO

Escopo em TINYPy – Variáveis e retorno

- Não existe declaração de variáveis em TINYPy: a primeira atribuição a uma variável a declara no escopo do bloco atual
 - Isso também implica que não existe ocultação de variáveis, e não existe redeclaração
- Uma função TINYPy retorna valores atribuindo a uma variável especial, com o mesmo nome da função
 - Forma mais simples é incluir esta variável na tabela de símbolos do corpo principal da função

