Compiladores – ASTs

Fabio Mascarenhas – 2017.2

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) ‘)’ << Fors>>

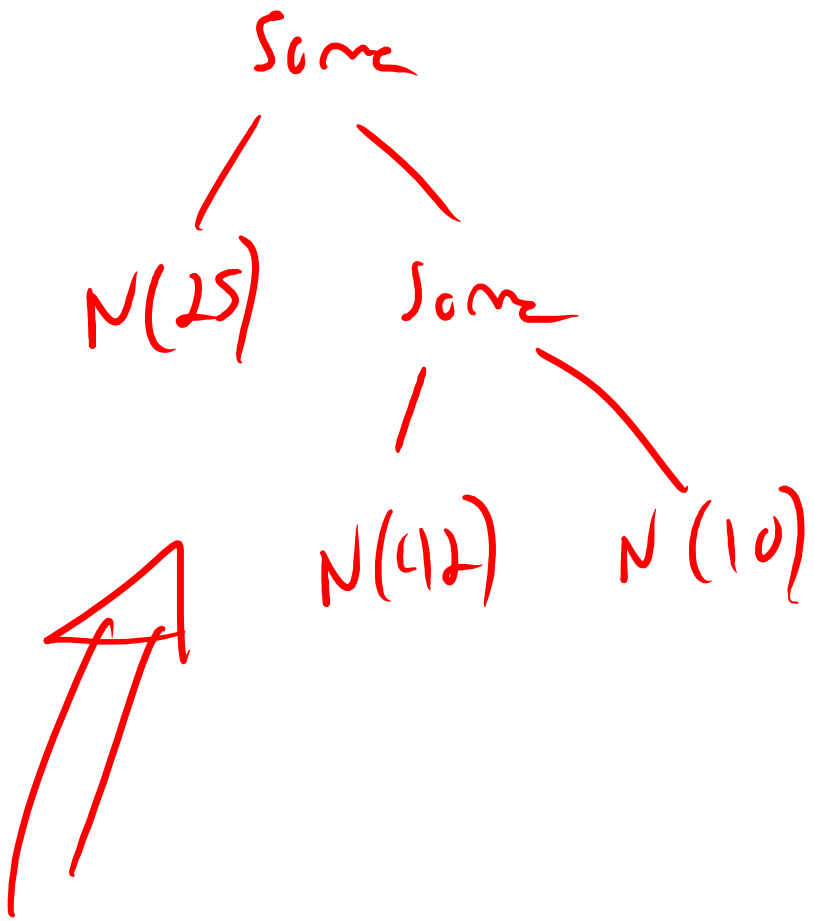
- Um analisador sintático bottom-up construiria a árvore sintática da próxima página

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 uma linguagem OO, podemos usar uma classe para cada tipo de nó → árvore heterogênea

- 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

data class Num(val value: String, val lin: Int) : Exp
data class Soma(val e1: Exp, val e2: Exp, val lin: Int): Exp
```

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>`

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