Construção de Compiladores

Parte 3

Construção da representação intermediária a partir da análise sintática

F.A. Vanini

IC – Unicamp

Klais Soluções

Ferramentas I

O analisador sintático ascendente é formado basicamente pelo algoritmo de análise sintática e por um conjunto de tabelas.

O algoritmo depende unicamente da variante do método de análise sintática utilizado.

As tabelas dependem não só do método como também da gramática.

Ferramentas II

Sendo assim, uma vez escolhido o método de análise sintática, a implementação do algoritmo pode ser a mesma para todas as gramáticas (que atendam às restrições impostas pelo método).

A construção manual dessas tabelas é trabalhosa e sujeita a erros.

A construção dessas tabelas através de ferramentas dedicadas não só é possível como também torna possível a “construção automática” do analisador sintática.

Ferramentas III

Essas ferramentas recebem como entrada a definição da gramática e produzem como saída as tabelas e uma implementação do algoritmo de análise sintática adaptada à gramática (tamanhos, nomes, etc...).

Elas produzem também a rotina básica para o *processo de tradução* desencadeado pela análise sintática.

Ferramentas IV

As ferramentas mais comuns são aquelas baseadas em variantes do método LR, pelo fato de serem mais eficientes e por impor poucas restrições à gramática.

Embora não sejam tão comuns, existem também ferramentas baseadas em métodos descendentes.

O Processo de Tradução I

O analisador sintático determina a estrutura da sentença de entrada (ou do programa).

A partir dessa estrutura é possível traduzir a sentença (ou o programa) para uma outra notação. Essa tradução constitui uma parte importante no processo de compilação.

Um exemplo I

O analisador sintático para a gramática de expressões usada no exemplo de análise descendente pode ser alterado para que à medida que a expressão for analisada, seja gerado um código para uma máquina bem simples.

- A gramática:

$$E \rightarrow T \{ + E \} \quad T \rightarrow F \{ * F \} \quad F \rightarrow (E) \mid a \mid b \mid c$$

- A máquina: a máquina alvo tem apenas 3 instruções:
  - *enter* <valor> :empilha um valor
  - *add* :retira dois valores da pilha, soma-os e empilha o resultado
  - *mult* :retira dois valores da pilha, multiplica-os e empilha o resultado

Um exemplo II

A expressão (a+b)*c seria traduzida para

enter a
enter b
add
enter c
mult

Ao tratar uma regra como **E → T { + E }** o analisador deve gera uma instrução “add” cada vez que a frase “+E” for reconhecida.

Análogamente, deve gerar uma instrução “mult” cada vez que a frase “ ***T** ” for reconhecida ao tratar a regra **T → F { * T }** .

Um exemplo III

```
procedimento E() {
    T;
    enquanto (simbolo lido == “+” ) {
        leia próximo simbolo;
        E;  escreva(“add”);
    }
 }

procedimento T() {
    F;
    enquanto (simbolo lido == “*” ) {
        leia próximo simbolo;
        T;  escreva(“mult”);
    }
 }
```

Um exemplo IV

```
procedimento F() {
   caso simbolo lido seja
      ´(´ : { leia proximo simbolo; E;
              se simbolo lido == ´)´ então leia próximo simbolo
                                     senão erro
            }
      ´a´, ´b´,´c´:  {
                        escreva(“enter “, simbolo lido);
                        leia proximo simbolo;
                     }
   outro: erro
}
```

O Processo de Tradução II

No caso da análise descendente, o evento importante é a redução e é nesse evento em que se baseia o processo de tradução.

O que se faz é associar a cada *regra* da gramática uma *ação* que é executada quando a redução correspondente for realizada pelo analisador sintático.

Um exemplo I

A mesma gramática para expressões usada para descrever o método LR, mostra como isso pode ser feito:

(1)	**E → E + T**	{ escreva(“add”); }
(2)	**E → T**	
(3)	**T → T * F**	{ escreva(“mult”);}
(4)	**T → F**	
(5)	**F → (E)**	
(6)	**F → id**	{ escreva(“enter “ + id); }

Um exemplo II

Uma ferramentas automatizada, nesse caso, geraria um procedimento semelhante ao mostrado abaixo, que seria chamado a cada redução:

```
procedimento exec_acao (regra_no) {
  caso regra_no seja:
      1:  { escreva("add"); }
      3:  { escreva("mult");}
      6:  { escreva("enter " + ident); }
}
```

Representação Intermediária I

A geração de código feita diretamente a partir da análise sintática, como mostrada até aqui, nem sempre é a forma mais conveniente:

- a ordem em que o código deve ser gerado nem sempre é aquela definida pela análise sintática
- eventualmente se deseja gerar código para uma máquina diferente

Representação Intermediária II

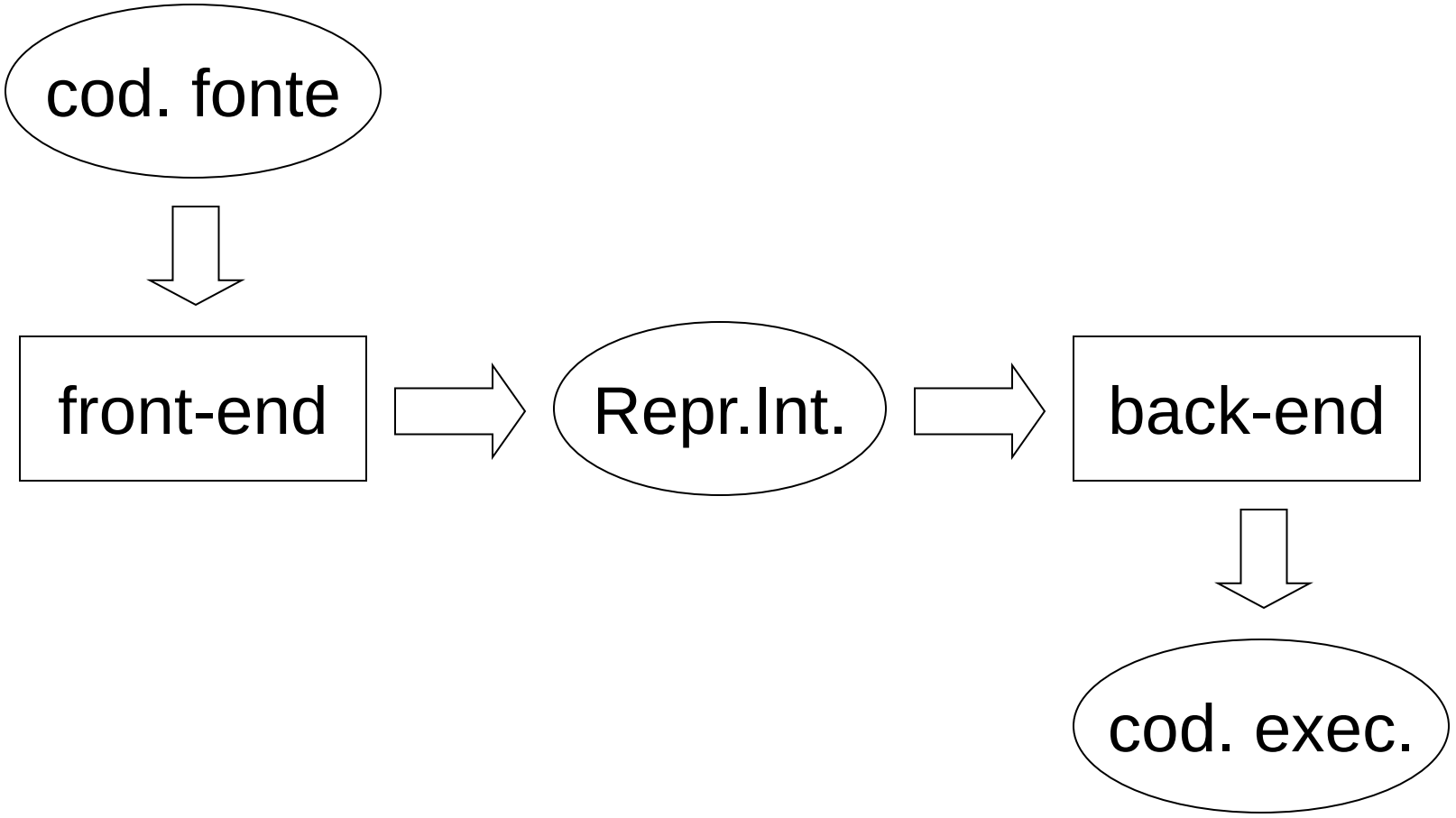
A maioria dos compiladores modernos traduz o programa fonte para uma representação intermediária, a partir da qual são feitas a análise semântica e a geração de código.

Além de tornar o compilador mais modular, esse esquema traz uma série de vantagens.

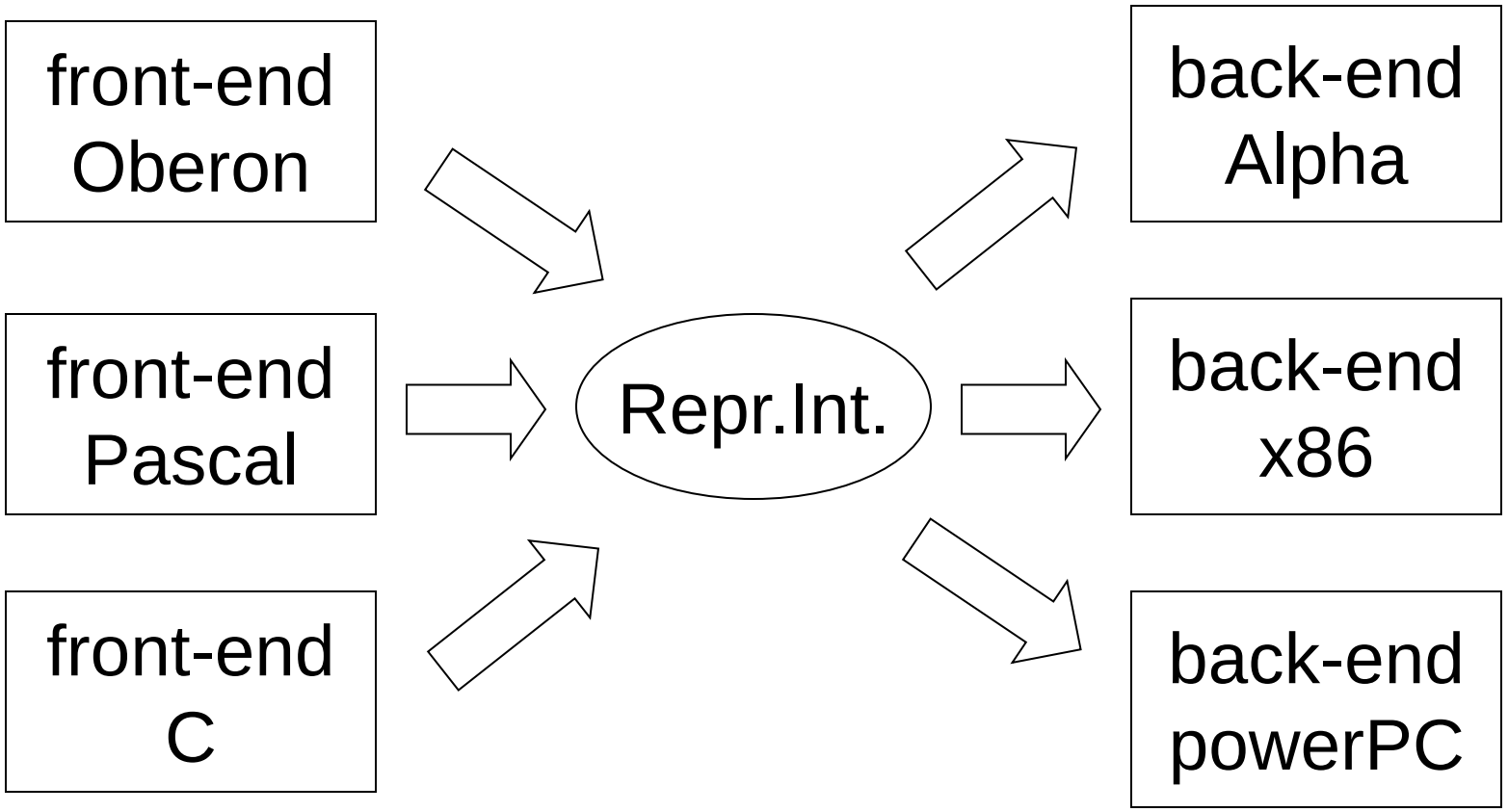
Representação Intermediária III

- a geração de código para outras máquinas reutiliza a análise sintática e a análise semântica.
- é possível fazer otimizações ao nível do código intermediário.
- uma mesma representação intermediária pode ser utilizada para linguagens diferentes, o que permite reutilizar os geradores de código.

Estrutura do Compilador I



Estrutura do Compilador II



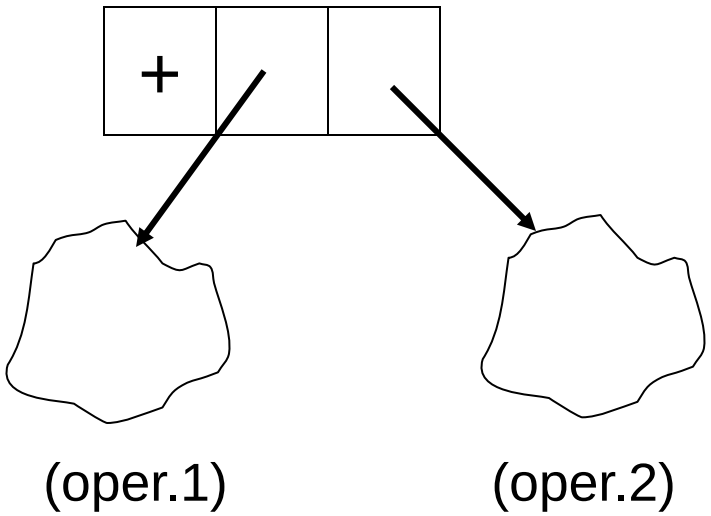
Representação Intermediária IV

Uma forma de construir a representação intermediária do programa é associar a cada operação da linguagem um nó ou objeto, que descreve a operação, fazendo eventuais referências às representações intermediárias das suas partes.

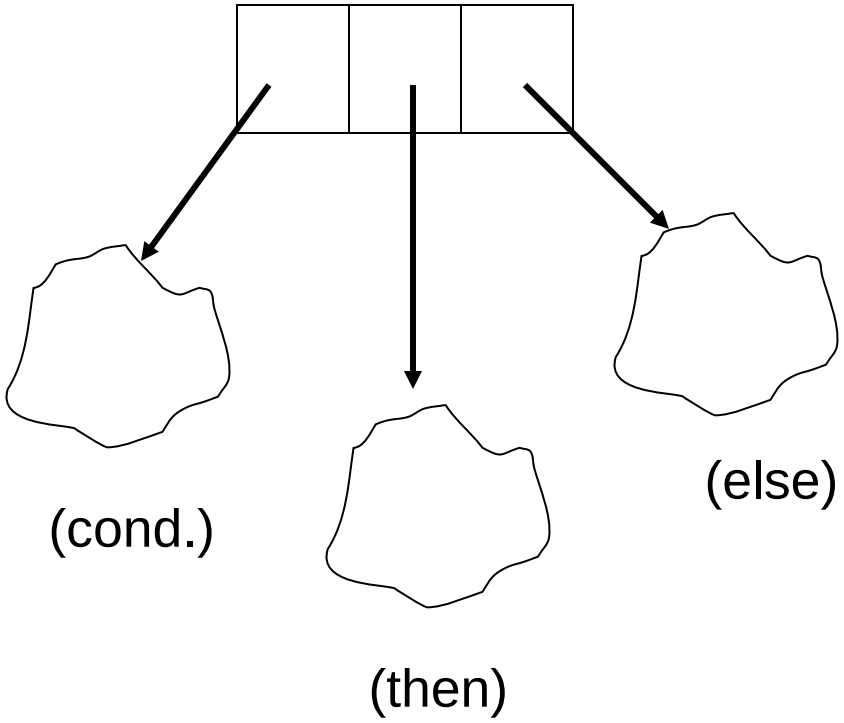
Representação Intermediária V

Exemplos

operação de soma



comando condicional



Hierarquia de Classes I

Cada operação da linguagem deve ser representada por um tipo específico de nó.

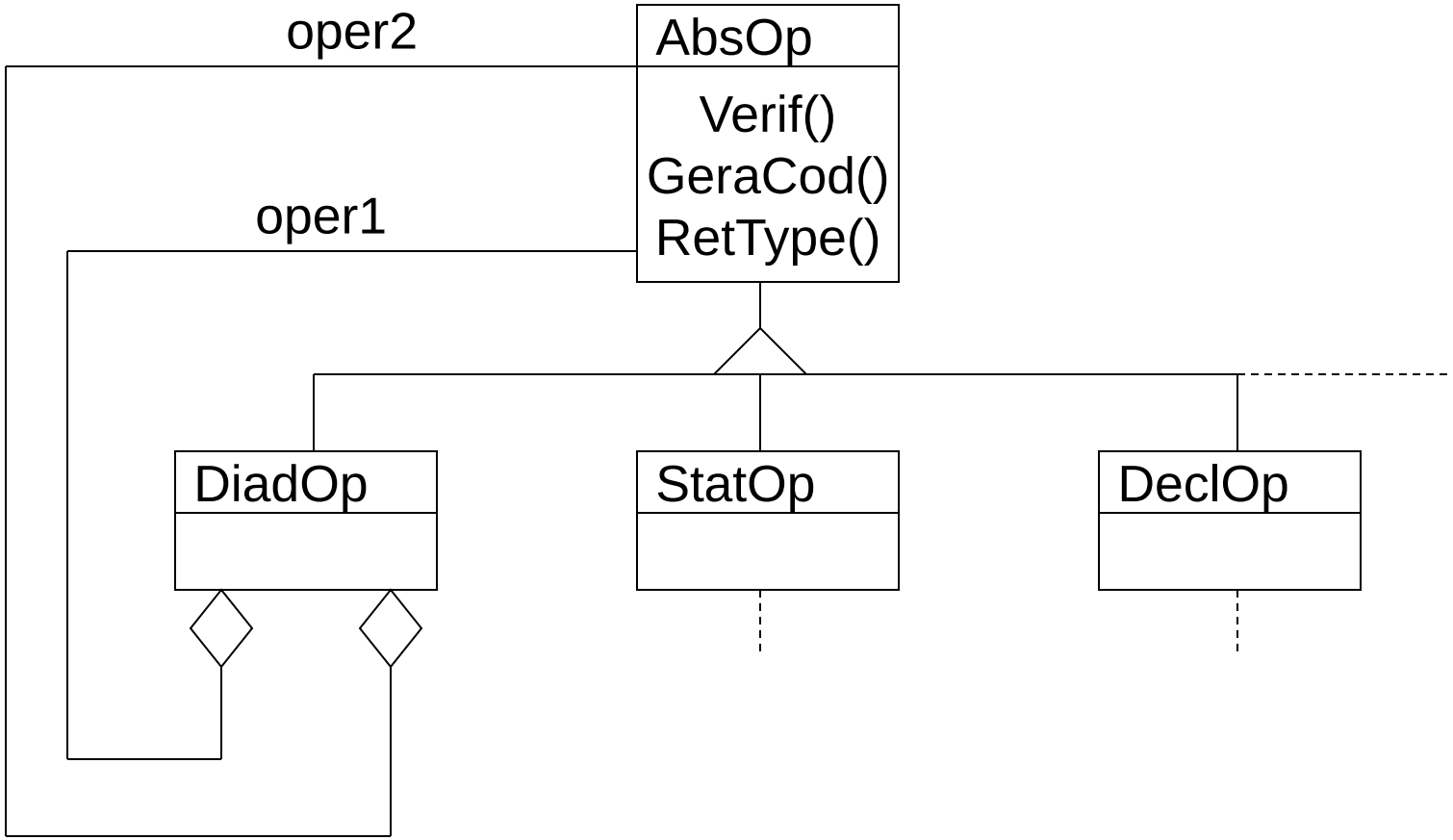
As operações realizadas sobre os nós serão basicamente as mesmas:

- Verificação semântica
- Geração de código
- Consulta ao tipo do valor retornado pela operação (usada na verificação semântica)

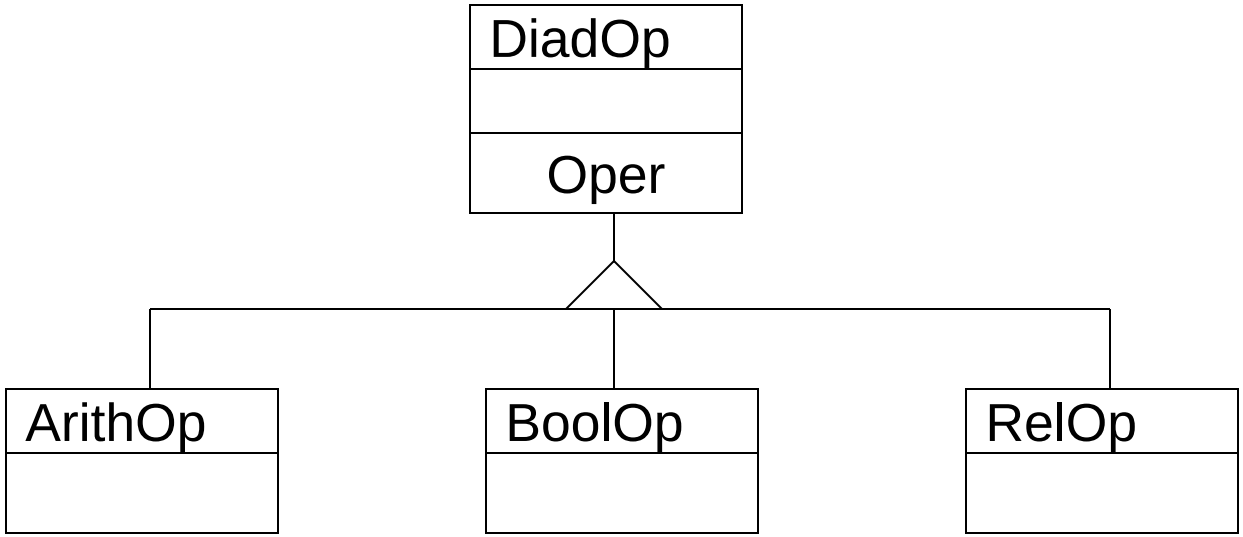
Hierarquia de Classes II

Ao implementar essas classes, é natural organizá-las numa hierarquia de classes de forma a aproveitar ao máximo as oportunidades de herança, fatorando o quanto antes os métodos comuns.

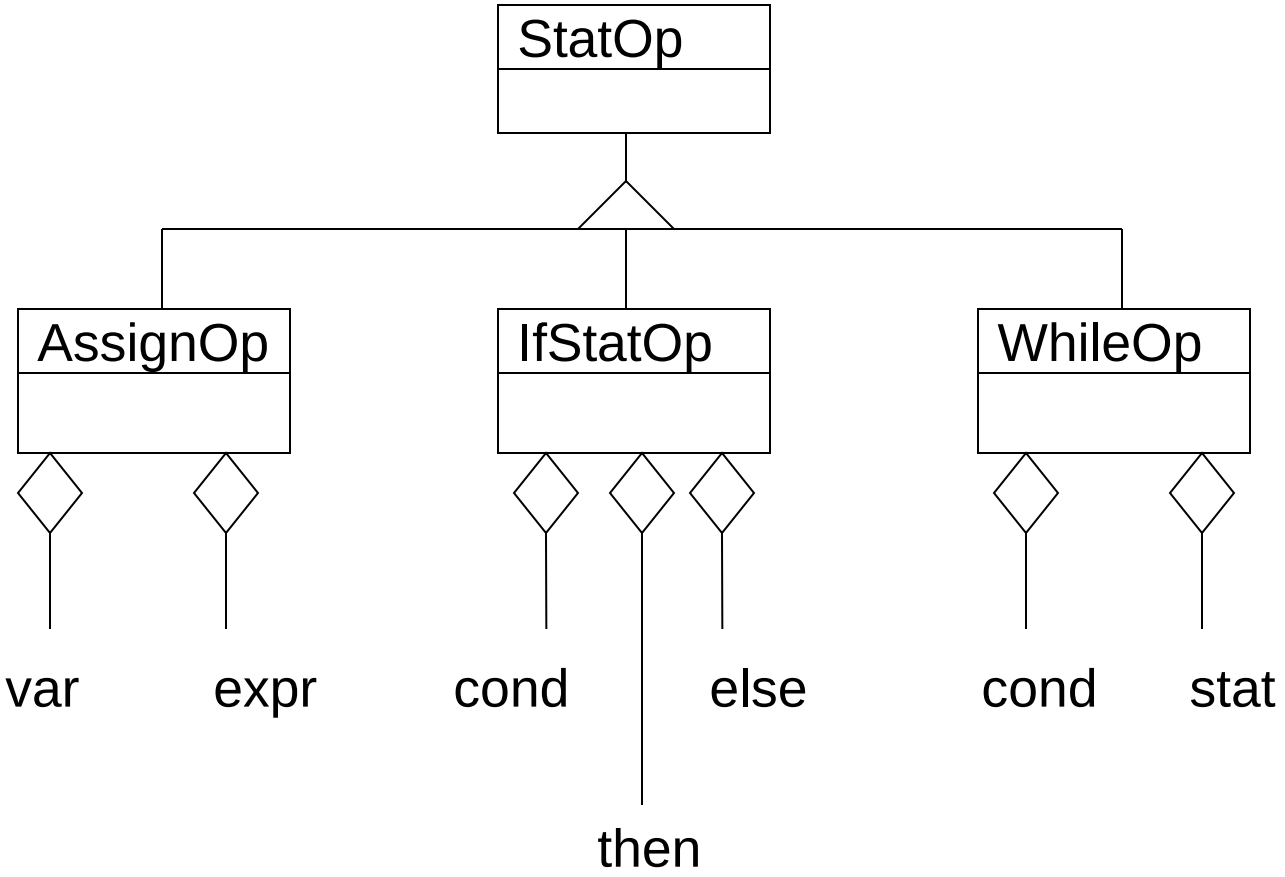
Hierarquia de Classes III



Hierarquia de Classes IV



Hierarquia de Classes IV



Hierarquia de Classes V

- A hierarquia de classes descrita é uma implementação do *Design Pattern* Composite (esse é o exemplo do livro clássico nesse assunto).
- O gerador de código pode ser implementado através do *Design Pattern* Visitor (esse também é o exemplo usado no mesmo livro para ilustrar o assunto). Nessa linha, diferentes geradores de código podem ser criados sem a necessidade de alteração nas classes da estrutura.

Verificação Semântica I

```
public abstract class DiadOp extends AbsOp{
  ...
  AbsOp opr1, opr2;
  ...
  boolean verif(){
    if( !opr1.verif() ||
        !opr2.verif() )  return false;
  t = this.retType();
  return ((opr1.retType() == t) &&
          (opr2.retType() == t) );
  }
}
```

Verificação Semântica II

```
public abstract class ArithOp extends DiadOp{
  ...
  Type retType() { return IntType; }
  ...
}

public abstract class BoolOp extends DiadOp {
  ...
  Type retType() { return Type.BoolType; }
  ...
}
```

Verificação Semântica III

```
public abstract class RelOp extends DiadOp{
  ...
  boolean verif(){
    if( !opr1.verif() ||
        !opr2.verif()  )  return false;
    return (opr1.retType()==opr2.retType);
  }
  ...
  Type retType(){ return Type.BoolType;}
  ...
}
```

Verificação Semântica IV

```
public class IfStatOp extends StatOp{
  ...
  AbsOp cond, thenStat, elseStat;
  ...
  boolean verif(){
    if( !cond.verif()   ||
      !thenStat.verif() ||
      !elseStat.verif()) return false;
    return cond.retType()==Type.BoolType;
  }
  Type retType(){ return Type.NoType; }
}
```

Tabela de Símbolos I

Durante a fase de verificação semântica, o que se faz é basicamente verificar se o *uso* dos elementos do programa, definidos pelo programador é *compatível* com a *definição* dos mesmos.

Para realizar esse tipo de verificação é necessário manter as informações relativas à declaração desse elementos numa *base de dados*, que é consultada durante a verificação.

Essa base de dados constitui a Tabela de Símbolos.

Tabela de Símbolos II

Durante a verificação semântica das declarações (de variáveis, funções, tipos, etc.), os símbolos (nomes) declarados são inseridos na Tabela de Símbolos.

Durante a verificação do uso desses nomes, a são feitas consultas à mesma.

A Tabela de Símbolos deve refletir as regras de visibilidade de nomes definidas na linguagem.

Tabela de Símbolos III

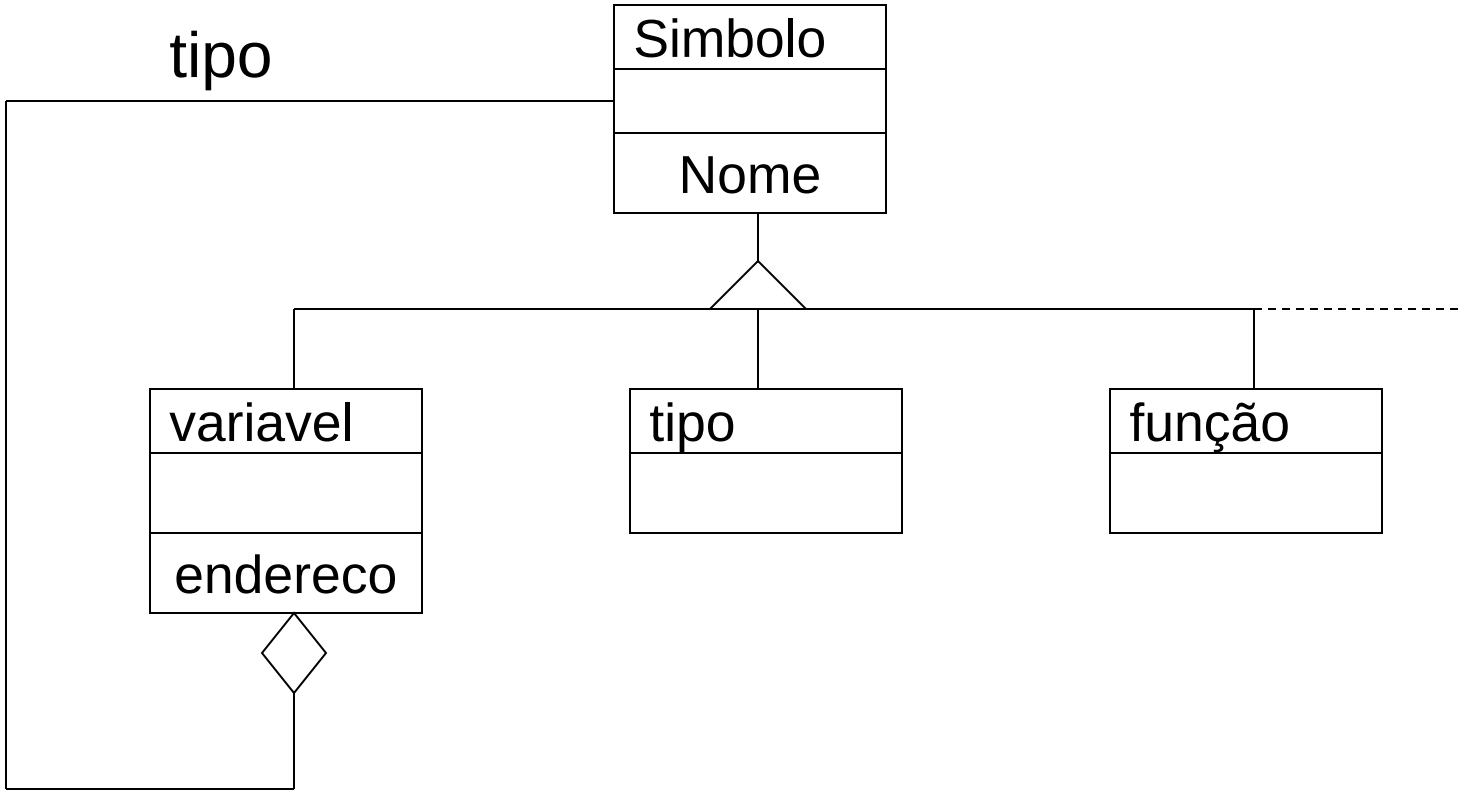
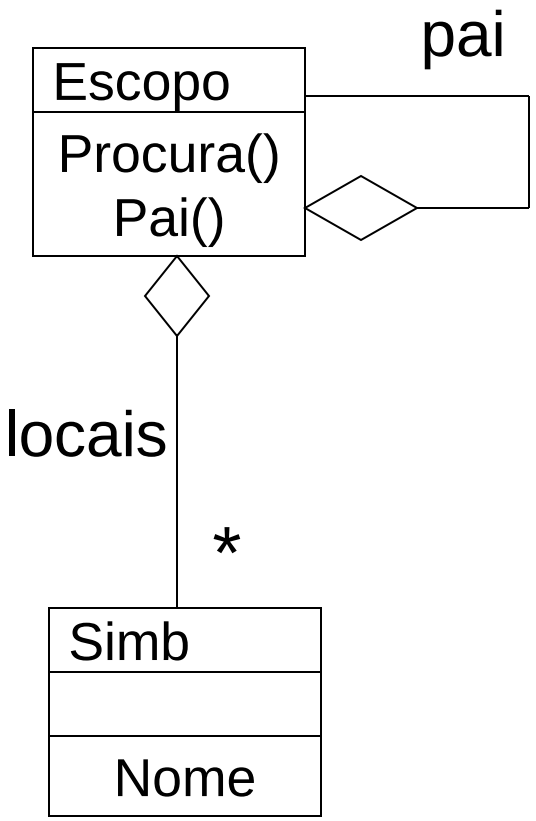


Tabela de Símbolos IV

A Tabela de Símbolos é uma Instância da classe Escopo.



Símbolos prédefinidos

```
public class Type extends Symb{
  // tipos pré-definidos
  static final Type NoType = new Type(...);
  static final Type IntType = new Type(...);
  static final Type BoolType = new Type(...)
  ...
}
```

Os símbolos prédefinidos são inseridos na Tabela de Símbolos antes de iniciar a compilação.

CUP
Gerador de analisador sintático LALR

Um exemplo: lista de expressões
(extraído de CUP User´s Manual)

- A gramática em notação BNF padrão:

```
expr_list ::= expr_list expr_part | expr_part
expr_part ::= expr ';'
expr ::= expr '+' expr
          | expr '-' expr
          | expr '*' expr
          | expr '/' expr
          | expr '%' expr
          | '(' expr ')'
          | '-' expr | number
```

Observação: a gramática acima é ambígua !

Um exemplo: lista de expressões

- Símbolos não terminais:

  `{ expr_list, expr_part  expr }`

- Nomes associados aos símbolos terminais:

  `SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD, NUMBER, LPAREN, and RPAREN`

Especificação para o CUP (1)

- Declarações iniciais :

```
// CUP specification for a simple expression
// evaluator (no actions)

import java_cup.runtime.*;
/* Preliminaries to set up and use the scanner. */
init with {: scanner.init(); :};
scan with {: return scanner.next_token(); :};
```

Especificação para o CUP (2)

- Definição dos símbolos terminais e não terminais:

```
/* Terminals (tokens returned by the scanner). */
terminal SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal UMINUS, LPAREN, RPAREN;
terminal Integer NUMBER;

/* Non terminals */
non terminal expr_list, expr_part;
non terminal Integer expr, term, factor;
```

Neste exemplo, os terminais não tem tipo. É possível associar tipos (classes) aos símbolos.

Especificação para o CUP (3)

- Precedência dos operadores (da menor para a maior)

```
/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;
```

Especificação para o CUP (4)

- A Gramática (sem nenhuma ação associada)

```
/* The grammar */
expr_list ::= expr_list expr_part | expr_part;
expr_part ::= expr SEMI;
expr ::= expr PLUS expr
       | expr MINUS expr
       | expr TIMES expr
       | expr DIVIDE expr
       | expr MOD expr
       | MINUS expr %prec UMINUS
       | LPAREN expr RPAREN | NUMBER
       ;
```

A sequência de regras associada a cada não terminal é finalizada com ‘;’.

Associando ações às regras

- Em CUP, as ações associadas às regras são escritas como comandos Java, delimitados por ‘{:’ e ‘:}’.
- Uma ação associada a uma regra pode retornar um valor. Nesse caso, o não terminal à esquerda dessa regra deve ser de um tipo compatível com esse valor.

Associando ações às regras

- Um exemplo:

```
non terminal expr_list, expr_part;
non terminal Integer expr;
...
expr_part ::= expr:e {: System.out.println("=" + e); :}
              SEMI
               ;
...
expr ::= expr:e1 PLUS expr:e2
       {: RESULT=new Integer(e1.intValue()+e2.intValue()); :}
        ...
       | LPAREN expr:e RPAREN
       {: RESULT = e; :} ;
```

Associando ações às regras

Nesse exemplo:

- ‘e’, ‘e1’ e ‘e2’ são rótulos que permitem às ações fazer referência aos diversos valores retornados pelos símbolos que compõem a regra.
- A palavra chave ‘RESULT’ indica o valor retornado pela regra.

Precedência e associatividade de operadores

- A diretiva ‘precedence’ define a precedência entre os operadores.
- Ela também pode ser usada para indicar a forma de associatividade entre os operadores.
- Um exemplo:

```
precedence left ADD, SUBTRACT;
precedence left TIMES, DIVIDE;
```

Ex.: 1+2+3 é reconhecida como (1+2)+3, eliminando uma possível ambiguidade da gramática.

Precedência Contextual

- A precedência de um operador pode ser forçada em função do contexto em que este aparece. Isso é feito através da diretiva %prec.
- Um exemplo:

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;
...
 expr ::= MINUS expr:e
        {: RESULT = new Integer(0 - e.intValue()); :}
        %prec UMINUS
```

Interface com o analisador léxico

- Para operar em conjunto com o analisador sintático gerado pelo CUP, o analisador léxico deve implementar a seguinte interface:

```
package java_cup.runtime;
public interface Scanner {
 public Symbol next_token()
     throws java.lang.Exception;
}
```

Referências

1. Design Patterns – elements of reusable object-oriented software, Gamma,E, et. al., Addison Wesley, 1995.
2. CUP User´s Manual, http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html