SBVCONC: Construção de Compiladores

Aula 07: Tratamento e Estratégias de Recuperação de Erros

Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Desenvolvimento do *Parser* da CPRL

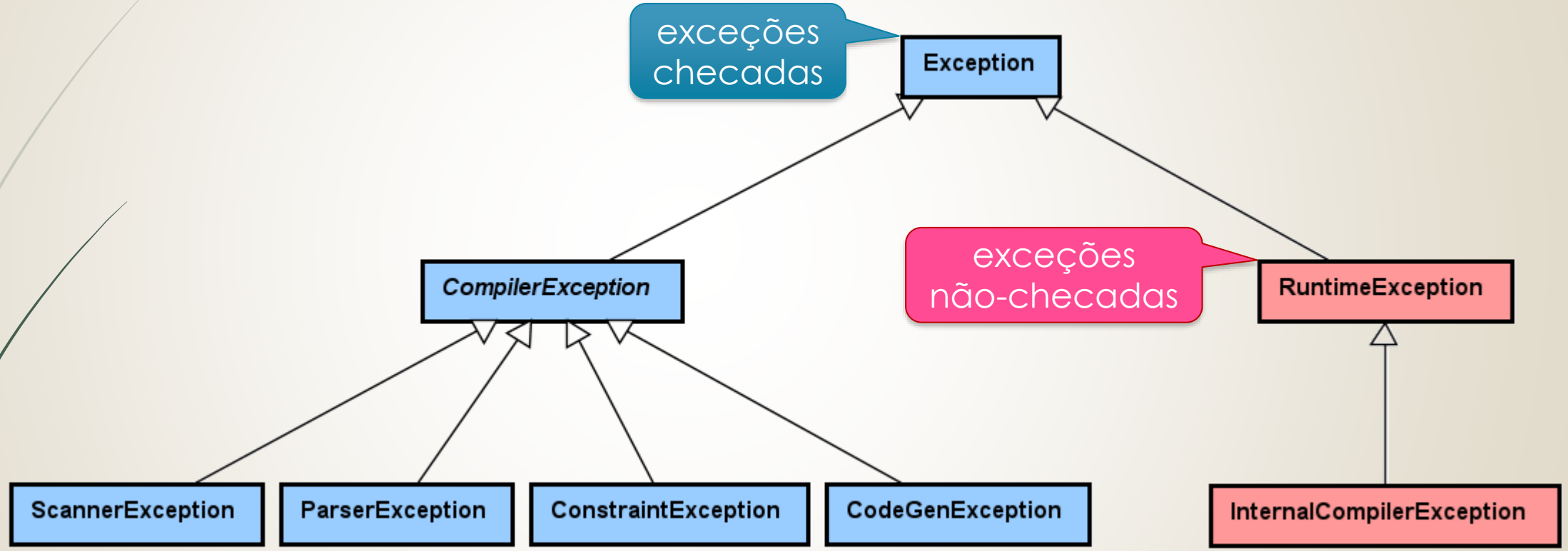
Versão 2: Tratamento e Recuperação de Erros (**Projeto 3**)

- Quando um compilador é integrado a um editor ou como parte de um ambiente integrado de desenvolvimento (IDE), pode ser que seja aceitável parar o processo de compilação após encontrar o primeiro erro e passar o controle ao editor;
- Em geral, mesmo quando integrado a um editor, um compilador deve tentar detectar e relatar tantos erros quanto possível.

Tipos de Erro de Compilação

- **Erros sintáticos:** gerados com base na gramática, por exemplo, tokens inválidos ou faltantes como a ausência de ponto de vírgula ou usar `"="` ao invés de `":="` para atribuição;
- **Erros de escopo:** violação de regras de escopo da linguagem, por exemplo, declarar dois identificadores com o mesmo nome dentro do mesmo escopo;
- **Erros de tipos:** violação de regras dos tipos da linguagem, por exemplo, a expressão que segue um `"if"` não ter o tipo `Boolean`;
- **Erros variados:** outros erros que não se encaixam nas categorias acima, por exemplo, funções não podem ter parâmetros `var`.

A Hierarquia de Exceções da CPRL



Exceções Checadas versus Exceções Não-Checadas

- Qualquer exceção que deriva da classe `Error` ou da classe `RuntimeException` é chamada de exceção não checada;
- Todas as outras exceções são chamadas de exceções checadas, ou seja, as exceções que herdam da classe `Exception` (com exceção, é claro, de `RuntimeException`);
- **Duas situações especiais:** se uma chamada ao um método que lança uma exceção checada é ou se uma exceção checada é explicitamente lançada, então o bloco que a contém precisa ou tratá-la localmente ou então o método em que ela é lançada deve declarar que essa exceção faz parte de sua lista de especificação de exceções;
- Exceções não checadas podem ser declaradas na lista de especificação de exceções ou podem ser tratadas, mas isso não é uma obrigação.

Manipulação de Erros versus Recuperação de Erros

- **Tratamento de Erros (*Error Handling*):** buscar por erros e relatá-los ao usuário;
- **Recuperação de Erros (*Error Recovery*):** o compilador tenta ressincronizar o seu estado e possivelmente o estado do fluxo de entrada de tokens de modo que a compilação possa continuar "normalmente";
- A proposta da recuperação de erros é encontrar a maior quantidade possível de erros em uma compilação única, com o objetivo de reportar quaisquer erros exatamente uma vez;
- Efetivamente, a recuperação de erros é extremamente difícil. Todos os erros reportados após o primeiro devem ser considerados suspeitos pelo programador.

Classe `ErrorHandler`

- Usada para que haja consistência no relatório de erros;
- Implementa o padrão de projeto *Singleton*;
- Termina o processo de compilação depois que uma quantidade fixa de erros for reportada;
- Para obter uma instância de `ErrorHandler` deve-se invocar:

```
ErrorHandler.getInstance();
```

Dois Métodos Chave da Classe `ErrorHandler`

```
/**
 * Retorna true caso algum erro tenha sido reportado.
 */
public boolean errorsExist()

/**
 * Reporta um erro. Para o processo de compilação se
 * uma quantidade máxima de erros for gerada.
 */
public void reportError( CompilerException e )
```

Estratégia Geral para o Tratamento de Erros

- Englobar o código de análise de cada regra dentro de um bloco `try/catch`;
- Quando um erro é detectado, o controle é transferido ao bloco `catch`;
- O bloco `catch` será responsável por:
  - Reportar o erro chamando os métodos apropriados no tratador de erros;
  - Pular os tokens até que algum token do conjunto $Follow$ do não-terminal representado pelo método de análise seja encontrado;
  - Retorna a execução ao método de análise que estava sendo executado.

O Método recover()

- O método `recover()` implementa a recuperação de erros ao pular os tokens até encontrar um que esteja no conjunto $Follow$ do não terminal definido pela regra.

```
/**
 * Avança o scanner até que o símbolo atual seja algum
 * dos símbolos especificados no array de seguidores.
 */
private void recover( Symbol[] followers ) throws IOException {
    scanner.advanceTo( followers );
}
```

Tratamento e Recuperação de Erros
Exemplo

- Considere a regra para uma `varDecl`:

```
varDecl = "var" identifiers ":" typeName ";" .
```

- Na CPRL, o conjunto Follow para a regra `varDecl` é:

```
{"const","var","type","procedure","function","begin"}
```

Tratamento e Recuperação de Erros
Exemplo

```
// varDecl = "var" identifiers ":" typeName ";" .
public void parseVarDecl() throws IOException {

    try {
        match( Symbol.varRW );
        List<Token> identifiers = parseIdentifiers();
        match( Symbol.colon );
        parseTypeName();
        match( Symbol.semicolon );
        for ( Token identifier: identifiers ) {
            idTable.add( identifier, IdType.variableId );
        }
    } catch ( ParserException e ) {
        ErrorHandler.getInstance().reportError( e );
        Symbol[] followers = {
            Symbol.constRW,     Symbol.varRW,      Symbol.typeRW,
            Symbol.procedureRW, Symbol.functionRW, Symbol.beginRW
        };
        recover( followers );
    }

}
```

Vamos fazer um pouquinho diferente 😊

Ó Método `parseVarDecl()` Reimplementado

```
// varDecl = "var" identifiers ":" typeName ";" .
public void parseVarDecl() throws IOException {

    try {

        ...

    } catch ( ParserException e ) {
        ErrorHandler.getInstance().reportError( e );
        recover( FOLLOW_SETS.get( "varDecl" ) );
    }

}
```

Assim 😉

Recuperação de Erros para o Método `parseStatement()`

- O método `parseStatement()` trata a regra:

```
statement = assignmentStmt | ifStmt | loopStmt | exitStmt
          | readStmt | writeStmt | writelnStmt
          | procedureCallStmt | returnStmt .
```

- A recuperação de erros para o método `parseStatement()` requere atenção especial quando o símbolo é um identificador, visto que um identificador pode não somente iniciar uma instrução, mas também pode aparecer em qualquer lugar desse tipo de construção;
- Considere, por exemplo, uma instrução de atribuição ou uma instrução de chamada de procedimento. Se avançarmos a um identificador, podemos estar no meio de uma instrução ao invés de estar no início da próxima instrução.

Recuperação de Erros para o Método `parseStatement()`

- Dado que a maioria dos erros relacionados aos identificadores estão relacionados à declarar ou referenciar um identificador de forma incorreta, assumiremos que esse é o caso e avançaremos ao próximo ponto e vírgula antes de implementar a recuperação de erros;
- O objetivo é que avançando ao próximo ponto e vírgula, com sorte, moveremos o scanner até o fim da instrução que contém o erro.

Recuperação de Erros para o Método `parseStatement()`

```
try {

    ...

} catch ( ParserException e ) {
    ErrorHandler.getInstance().reportError( e );
    scanner.advanceTo( Symbol.semicolon );
    recover( FOLLOW_SETS.get( "statement" ) );
}
```

Implementando a Recuperação de Erros

- Nem todos os métodos necessitarão de um bloco `try/catch` para a recuperação de erros nesse estágio de desenvolvimento do `parser`;
- **Exemplo:**

```
public void parseInitialDecls() throws IOException {
    while ( scanner.getSymbol().isInitialDeclStarter() ) {
        parseInitialDecl();
    }
}
```

- O método `match()` lança uma `ParseException` quando um erro é detectado, ele não implementa recuperação de erros;
- Quaisquer métodos de análise que chamam o método `match()` necessitarão um bloco `try/catch` para a recuperação de erros.

Implementando a Recuperação de Erros

- Adicionalmente, o método `parseVariableExpr()` e o método `add()` da classe `IdTable` pode lançar uma `ParserException`.
- Os métodos `match()`, `add()` e `parseVariableExpr()` são os únicos três métodos que lançam uma `ParserException` de volta ao método chamador, sendo assim, qualquer método que usa um desses três métodos terão que ter um bloco `try/catch`.

Estratégias Adicionais para a Recuperação de Erros

- Após reportar o erro, trocar o token atual com o token que deveria ser permitido naquele ponto do processo de análise;
- **Exemplos:**
  - Trocar "=" por ":=" quando estiver analisando uma instrução de atribuição da linguagem CPRL;
  - Trocar "=" por "==" quando se espera um operador relacional na linguagem Java.

Estratégias Adicionais para a Recuperação de Erros

- Após reportar o erro, inserir um novo token à frente do que gerou o erro;
- **Exemplos:**
  - Quando se está analisando uma instrução *exit*, após verificar a correspondência (match) do token `"exit"`, se o símbolo encontrado estiver no conjunto $First$ de `expression`, inserir um token `"when"` e continuar a análise;
  - Ao se analisar uma expressão, se um `")"` for esperado, mas um `";"` for encontrado como próximo símbolo, inserir um `")"` esperando que um `";"` provavelmente encerrará a instrução.

Estratégias Adicionais para a Recuperação de Erros

Exemplo no Método `parseAssignmentStmt()`

```
try {...

...

    // match( Symbol.assign );
    try {

        match( Symbol.assign );

    } catch ( ParserException e ) {

        if ( scanner.getSymbol() == Symbol.equals ) {
            ErrorHandler.getInstance().reportError( e );
            matchCurrentSymbol();  // tratar "=" como ":="
                                   // nesse contexto
        } else {
            throw e;
        }

    }

    ...

}...
```

Ao invés de simplesmente invocar `match( Symbol.assign )`, usamos um bloco `try/catch` aninhado que fará o tratamento de `"="` como `":="`

Essa estratégia está implementada no parser do Projeto 2!

Bibliografia

MOORE JR., J. I. **Introduction to Compiler Design: an Object Oriented Approach Using Java**. 2. ed. [s.l.]:SoftMoore Consulting, 2020. 284 p.

AHO, A. V.; LAM, M. S.; SETHI, R. ULLMAN, J. D. **Compiladores: Princípios, Técnicas e Ferramentas**. 2. ed. São Paulo: Pearson, 2008. 634 p.

COOPER, K. D.; TORCZON, L. **Construindo Compiladores**. 2. ed. Rio de Janeiro: Campus Elsevier, 2014. 656 p.

JOSÉ NETO, J. **Introdução à Compilação**. São Paulo: Elsevier, 2016. 307 p.

SANTOS, P. R.; LANGOLOIS, T. **Compiladores: da teoria à prática**. Rio de Janeiro: LTC, 2018. 341 p.