SBVCONC: Construção de Compiladores

Aula 09: Análise de Restrições (*Constraint Analysis*)

Análise Contextual (*Contextual Analysis*)
Análise da Semântica Estática (*Analisys of Static Semantics*)

Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

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

A Especificação de uma Linguagem de Programação

- Sintaxe (forma):
  - Símbolos básicos da linguagem (*tokens*);
  - Estruturas de símbolos que são permitidas para se formar programas;
  - Especificada por uma gramática livre de contexto;
- Restrições Contextuais:
  - Regras e restrições do programa que não podem ser especificadas em uma gramática livre de contexto;
  - Consistem primariamente em regras de tipos e de escopo;
- Semântica (significado):
  - Comportamento do programa quando é executado em uma máquina;
  - Normalmente especificada informalmente.

Análise Sintática versus Análise de Restrições

- A análise sintática verifica se um programa está em conformidade com a sintaxe formal da linguagem definida por uma gramática livre de contexto;
- A análise sintática é executada pelo *parser*;
- A análise de restrições verifica se um programa está em conformidade com regras e requisitos adicionais da linguagem, sendo expressa informalmente;
- A análise de restrições é executada parcialmente pelo *parser* usando as classes auxiliares `IdTable`, `LoopContext` e `SubprogramContext` e parcialmente pelas ASTs nos métodos `checkConstraints()`.

Categorias de Restrições

- As restrições podem ser classificadas em três categorias gerais:
  - **Regras de Escopo:** são as regras associadas às declarações e às ocorrências aplicadas dos identificadores;
  - **Regras de Tipo:** são associadas com os tipos das expressões e seus usos em determinados contextos;
  - **Regras Variadas:** são as restrições da linguagem que não se encaixam nas duas classificações anteriores. Algumas dessas regras representam erros internos do compilador que podem ter ocorrido durante o processo de análise sintática.

Regras de Escopo na CPRL

- As regras de escopo da CPRL compreendem:
  - Todos os identificadores definidos pelo usuário (constantes, variáveis, nomes de tipos, nomes de subprogramas etc.) devem ser declarados. Quando uma ocorrência aplicada de um identificador é encontrada, temos que ser capazes de descobrir sua respectiva declaração e associá-la ao uso do identificador;
  - Todos os identificadores que aparecem nas declarações devem ser únicos dentro de seus escopos. Em outras palavras, o mesmo identificador não pode ser usado em duas declarações diferentes no mesmo escopo;
  - A CPRL tem o que é chamado de *flat block structure*, onde as declarações são de escopo global ou local aos subprogramas.

Análise de Escopo (*Identification*)

- A análise de escopo, ou simplesmente *Identification* (Identificação), é o processo de verificação das regras de escopo;
- Para a CPRL, a análise de escopo é implementada dentro do *parser* usando a classe `IdTable`;
- A classe `IdTable` é capaz de manipular escopos aninhados de dois níveis, como requerido pela CPRL, mas pode ser facilmente estendida para permitir o gerenciamento de níveis arbitrários de aninhamento de escopos, assim como na linguagem Java.

Análise de Escopo Usando a Classe `IdTable`

- Quando um identificador é declarado, o *parser* tentará adicionar uma referência à sua declaração na `IdTable` dentro do escopo atual/corrente;
  - Lança uma exceção caso uma declaração com o mesmo nome (mesmo texto do *token*) tiver sido adicionada previamente ao escopo corrente;
- Quando uma ocorrência aplicada de um identificador é encontrada, por exemplo, em uma instrução, o *parser* irá:
  - Verificar se o identificador foi declarado;
  - Armazenar uma referência ao identificador da declaração como parte da AST no ponto em que o identificador foi usado.

Tipagem Estática (*Static Typing*)

- A CPRL é uma linguagem de tipagem estática:
  - Todas as variáveis e expressões têm um tipo;
  - A compatibilidade entre tipos é uma propriedade estática, ou seja, pode ser determinada pelo compilador;
- As regras de tipo para a CPRL definem como e quando certos tipos podem ser usados;
- Definiremos as regras de tipos em quaisquer contextos onde uma variável ou expressão podem aparecer.

Exemplos de Regras de Tipos na CPRL

- Para uma instrução de atribuição, o tipo da variável do lado esquerdo do símbolo de atribuição devem ser do mesmo tipo da expressão do lado direito;
- Note que algumas linguagens não requerem essa igualdade (mesmo tipo), verificando para isso se os tipos tem compatibilidade de atribuição. Por exemplo, em C é perfeitamente aceitável atribuir um caractere à uma variável inteira;
- Para expressões de negação, o operando deve ter o tipo `Integer` e o resultado de uma expressão de negação é do tipo `Integer`.

Análise de Restrições

- A análise de restrições é o processo de verificar se todas as regras de restrição foram satisfeitas;
- Para a CPRL, a maioria das regras variadas e de tipo são verificadas ao se usar o método `checkConstraints()` das classes da AST;
- Mesmo as classes da AST que não possuem restrições associadas implementarão o método `checkConstraints()` caso elas tenham referências à outros objetos da AST. Para isso, elas simplesmente invocarão o método `checkConstraints()` nesses objetos.

Verificação de Restrições para a Classe Program

```
@Override
public void checkConstraints() {

    assert declPart != null : "declPart should never be null.";
    assert stmtPart != null : "stmtPart should never be null.";

    declPart.checkConstraints();
    stmtPart.checkConstraints();

}
```

Verificação de Restrições para a Classe `DeclarativePart`

```
@Override
public void checkConstraints() {

    for ( InitialDecl decl : initialDecls ) {
        decl.checkConstraints();
    }

    for ( SubprogramDecl decl : subprogDecls ) {
        decl.checkConstraints();
    }

}
```

Verificação de Restrições para a Classe `StatementPart`

```
@Override
public void checkConstraints() {

    for ( Statement stmt : statements ) {
        stmt.checkConstraints();
    }

}
```

Regras de Restrição para a CPRL/0

- Expressões de Adição e Multiplicação:
  - **Regra de Tipo:** ambos os operandos devem ser do tipo `Integer`;
  - **Regra Variada:** o resultado tem que ser do tipo `Integer`;
- Instrução de Atribuição:
  - **Regra de Tipo:** a variável (lado esquerdo do operador de atribuição) e a expressão (lado direito) devem ter o mesmo tipo;
- Instrução **`exit`**:
  - **Regra de Tipo:** se uma expressão `when` existir, o seu tipo deve ser `Boolean`;
  - **Regra Variada:** a instrução `exit` deve estar aninhada dentro de uma instrução de laço, o que é tratado pelo *parser* usando `LoopContext`.

Regras de Restrição para a CPRL/0

- Instrução **`if`**:
  - **Regra de Tipo:** a expressão deve ser do tipo `Boolean`;
  - **Regra de Tipo:** as expressões para quaisquer cláusulas `elsif` devem ser do tipo `Boolean`;
- Instrução **`read`**:
  - **Regra de Tipo:** a variável deve ser do tipo `Integer` ou do tipo `Char`;
- Expressão Lógica:
  - **Regra de Tipo:** ambos os operandos devem ser do tipo `Boolean`;
  - **Regra Variada:** o resultado tem que ser do tipo `Boolean`.

Regras de Restrição para a CPRL/0

- Instrução **`loop`**:
  - **Regra de Tipo:** se uma expressão `while` existir, ela tem que ser do tipo `Boolean`;
- Expressão de Negação (*Negation*):
  - **Regra de Tipo:** o operando tem que ser do tipo `Integer`;
  - **Regra Variada:** o resultado tem que ser do tipo `Integer`;
- Expressão Não (*Not*):
  - **Regra de Tipo:** o operando tem que ser do tipo `Boolean`;
  - **Regra Variada:** o resultado tem que ser do tipo `Boolean`.

Regras de Restrição para a CPRL/0

- Expressão Relacional:
  - **Regra de Tipo:** ambos os operandos devem ter o mesmo tipo;
  - **Regra de Tipo:** apenas os tipos escalares, `Integer`, `Char` ou `Boolean`, são permitidos como operandos. Na CPRL, não é permitido que ambos os operandos sejam arrays ou literais de String;
  - **Regra Variada:** o resultado tem que ser do tipo `Boolean`;
- Declaração de Variáveis e de Variável Única:
  - **Regra de Tipo:** o tipo deve ser `Integer`, `Boolean`, `Char` ou um tipo de array definido pelo usuário.

Regras de Restrição para a CPRL/0

- Declaração e Valor de Constantes:
  - **Regra Variada:** se o valor do literal for do tipo `Integer`, então é necessário que o mesmo possa ser convertido em um inteiro na CPRL *Virtual Machine*. Em outras palavras, verificar se `Integer.parseInt()` não falha. Se a verificação falhar para uma declaração de constante, então configure o valor do literal como um valor válido para um `Integer` de modo a prevenir mensagens de erros adicionais toda vez que uma declaração da constante for utilizada.
- Instrução **`write`**:
  - **Regra Variada:** para uma instrução `write`, mas não para `writeln`, é necessário que exista pelo menos uma expressão.

Regras de Restrição para Subprogramas

- Instrução **`return`**:
  - **Regra de Tipo:** se a instrução retorna o valor de uma função, então o tipo da expressão que será retornada deve ser do mesmo tipo do retorno da função;
  - **Regra Variada:** se a instrução `return` retorna um valor, então ela deve estar aninhada à declaração de uma função;
  - **Regra Variada:** se a instrução `return` está aninhada a uma função, então ela deve retornar um valor;
  - **Regra Variada:** a instrução `return` deve estar aninhada a um subprograma, o que é tratado pelo *parser* usando `SubprogramContext`.

Regras de Restrição para Subprogramas

- Declaração de Função:
  - **Regra Variada:** não pode haver nenhuma `var` nos parâmetros;
  - **Regra Variada:** é necessário que haja pelo menos uma instrução de retorno;
- Chamada de Subprograma, tanto para procedimentos quanto para funções:
  - **Regra de Tipo:** a quantidade de argumentos (*actual parameters*) precisa ser a mesma da quantidade de parâmetros formais e cada par deve ter o mesmo tipo;
- Chamada de Procedimento:
  - **Regra Variada:** se um parâmetro formal é um parâmetro `var`, então o argumento deve ser um valor nomeado, não uma expressão arbitrária.

Regras de Restrição para Arrays

- Declaração de Tipo de Array:
  - **Regra de Tipo:** o valor da constante que especifica a quantidade de itens de um array deve ser do tipo `Integer` e o valor associado deve ser um número positivo;
- Variáveis e Valores Nomeados:
  - **Regra de Tipo:** Cada expressão de índice deve ser do tipo `Integer`;
  - **Regra Variada:** O uso de expressões nos índices são permitidas apenas em variáveis de tipos de array.

Verificação de Restrições para a Classe `AssignmentStmt`

```
@Override
public void checkConstraints() {

    try {

        expr.checkConstraints();
        variable.checkConstraints();

        if ( !matchTypes( variable.getType(), expr.getType() ) ) {
            String errorMsg = "Type mismatch for assignment statement.";
            throw error( assignPosition, errorMsg );
        }

    } catch ( ConstraintException e ) {
        ErrorHandler.getInstance().reportError( e );
    }

}
```

Verificação de Restrições para a Classe `NegationExpr`

```
@Override
public void checkConstraints() {

    try {

        Expression operand = getOperand();
        operand.checkConstraints();

        // os operadores unários +/- podem ser aplicados somente à expressões do tipo Integer
        if ( operand.getType() != Type.Integer ) { // pode usar matchTypes também!
            String errorMsg = "Expression ...";
            throw error( operand.getPosition(), errorMsg );
        }

    } catch ( ConstraintException e ) {
        ErrorHandler.getInstance().reportError( e );
    }

}
```

Projeto 5: Implementação da Análise de Restrições do Compilador da CPRL

- Implemente todos os métodos `checkConstraints()` das classes da AST;
- Verifique no projeto o arquivo "Visão Geral das Classes do Projeto.txt" em que as classes que precisam ser modificadas estão listadas;
- As implementações que devem ser feitas estão explicadas em comentários dentro dos métodos `checkConstraints()` que precisam ser implementados;
- Há várias classes com a implementação pronta e que podem ser usadas como base para a implementação das que precisam ser complementadas.

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.