Instituto Tecnológico de Aeronáutica

Curso de Engenharia da Computação

Disciplina CES-41: Compiladores

Relatório do Laboratório 4

Vitor Pimenta dos Reis Arruda

José Luciano de Morais Neto

Professor Fábio Carneiro Mokarzel

São José dos Campos

09/06/2019

Sumário

1 Introdução

Este trabalho intenciona a escrita e o teste de um analisador semântico para a linguagem de programação COMP-ITA 2019 [1]. Implementa-se uma solução que processa um programa escrito na tal linguagem, captura sua árvore sintática, opera sobre ela e imprime (i) a tabela de símbolos do programa e (ii) todos os erros semânticos encontrados, segundo uma listagem de regras semânticas especificadas pela linguagem.

Para o desenvolvimento, usou-se a linguagem de programação **typescript** com o analisador léxico **Moo** [3] e o analisador sintático **Nearley** [4]. A análise semântica foi empreendida por pós-processamento da árvore sintática, completamente usando a linguagem de programação mencionada, sem ajuda de bibliotecas externas.

As seções seguintes se estruturam de maneira a apresentar, em linhas gerais, a estratégia adotada para a análise e os resultados obtidos com a execução sobre programas-teste.

2 Tabela de símbolos

Definiu-se um “escopo” para cada função e um escopo “global” acima dos primeiros. Quando um símbolo é inserido, deve-se especificar em qual escopo ele estará. Também, quando um símbolo é pesquisado, especifica-se o escopo pela qual a busca iniciará e, caso o símbolo não esteja no tal escopo, o escopo global é pesquisado.

A tabela foi representada por uma classe, cuja interface é como ilustrado pela Listagem 1:

```
export interface SymbolTable {

    createLocalScope(functionObj: IFunction): LocalScope

    registerSymbol(inScope: IScope, symbol: ISymbol): void

    /**
     *
     * @param fromScope Scope to begin the search. Also searches the global scope later,
     * if not found in local scope
     * @param symbolName
     * @param dontFollowScopes If `true`, will search *only* in the requested scope
     */
    getSymbolEntry(
        fromScope: IScope,
```

```
        symbolName: string,
        dontFollowScopes: boolean
    ): ISymbol | null

    getGlobalScope(): GlobalScope | null

    getLocalScope(functionName: string): LocalScope | null

}
```

Listagem 1: Interface da tabela de símbolos

Observa-se que a tabela não armazena dados que requeiram processamento posterior para serem resolvidos – como contagem de referências a símbolos e *flags* para indicar se foram inicializados ou não. A tabela meramente possibilita encontrar, a partir de um escopo e um nome de símbolo, o símbolo correspondente (identificador ou função).

3 Regras semânticas

Todas as regras semânticas definidas pela especificação da linguagem COMPITA-2019 foram representadas por classes. A Listagem 2 enumera todas as classes necessárias para completa cobertura de todas as regras pretendidas (o corpo de cada classe foi removido para enfatizar somente o nome de cada uma):

```
/**
* There must be one, and only one, main function
*/
export class UniqueMainFunction { }

/**
* If a variable is referenced somewhere, it must have
* been declared
*/
export class DeclareBeforeUse { }

/**
* A function call (either a CALL or inside an expression)
* cannot be done if the symbol is not a function
* (for example, an int is not callable)
*/
export class IfCalledThenIsFunction { }

/**
* CALL x() cannot have `x` being a non-void function
*/
export class CallStatementMustReturnVoid { }
```

```
/**
* Forbidden to declare `void x`
*/
export class NoVoidIdentifier { }

/**
* Operators (+, *, &&, ||, ~, !) must have operands
* whose types are acceptable
*/
export class OperandsCompatibleWithOperators { }

/**
* Forbidden to declare `int x[0]`
*/
export class PositiveVectorDimensions { }

/**
* If a variable is declared, it must be initialized
* (i.e. be the target of an assignment or a Read)
* and must be referenced (read from)
*/
export class IfDeclaredThenMustInitializeAndReference { }

/**
* If `x <- y`, the type of `y` must be assignable
* to the type of `x`
*/
export class AssignmentTypeCompatibility { }

/**
* If declared `int x[1, 2]` then must access it
* as `x[i1, i2]` (i.e. there must be 2 subscripts)
*/
export class IndexingDimensionsMustMatch { }

/**
* Branching commands must have a logical expression
* as their 'condition'
*/
export class IfWhileDoForMustHaveLogicalExpressions { }

/**
* A FOR command must be initialized by an int scalar
* variable
*/
export class ForMustBeInitializedByScalar { }
```

```
/**
* If `x` is used to initialize a FOR, then
* its increment variable must also be `x`
*/
export class ForInitializerMustMatchIncrement { }

/**
* Forbidden to use `x[expression]` if the resolved
* type of `expression` is not assignable to int
* (int or char)
*/
export class MustIndexWithIntLikeExpressions { }

/**
* Inside an expression, no subexpression may have
* a void type. For example, if `void x() {...}`
* then it is forbidden to have `1 + x()` since
* the type of `x` invocation is void
*/
export class ExpressionDoesNotAdmitVoidCalls { }

/**
* No variable or function or parameter may have the
* same name as the program
*/
export class NoClashWithProgramName { }

/**
* A function name cannot be passed as argument to
* a function, or used as expression anywhere
*/
export class NoFunctionPointers { }

/**
* If a function is declared with `n` arguments,
* are calls to it must specify `n` arguments
*/
export class ArgumentCountsMustMatch { }

/**
* For each argument in a function call, its
* type must be compatible with the type
* specified in the function signature
*/
export class ArgumentTypesMustBeCompatible { }

/**
* The type of a return statement must be assignable
```

```
* to the function's return type
*/
export class ReturnStatementMustMatchFunctionType { }

/**
* Either direct or indirect recursion is
* forbidden
*/
export class RecursiveCallsAreNotSupported { }
```

Listagem 2: Todas as regras semânticas da linguagem, representadas por classes

4 Erros semânticos

Assim como se definiram as regras semânticas, também todos os possíveis erros semânticos (oriundos de transgressões às supracitadas regras) foram representados por classes, como na Listagem 3. Cada erro armazena o nó da árvore sintática em que ele ocorreu, e é capaz de informar a posição no código-fonte onde há o problema.

```
export interface CodeLocalization {
    line: number
    col: number
}

export interface CodeRange {
    begin: CodeLocalization
    end: CodeLocalization
}

export interface SemanticalError {
    message: string
    where: ASTNode
    localize(backmap: Backmap): CodeRange
}

export class DuplicateDeclaration implements SemanticalError { }
export class MissingMainFunction implements SemanticalError { }
export class Undeclared implements SemanticalError { }
export class NotAFunction implements SemanticalError { }
export class NonVoidCall implements SemanticalError { }
export class VoidIdentifier implements SemanticalError { }
export class IncompatibleType implements SemanticalError { }
export class NonPositiveVectorDimension implements SemanticalError { }
export class NotInitialized implements SemanticalError { }
export class NotReferenced implements SemanticalError { }
export class MismatchingDimensionality implements SemanticalError { }
```

```
export class UnexpectedForInitialization implements SemanticalError { }
export class UnrelatedForIncrement implements SemanticalError { }
export class WrongIndexingType implements SemanticalError { }
export class VoidInExpression implements SemanticalError { }
export class SameNameAsProgram implements SemanticalError { }
export class FunctionPointerReference implements SemanticalError { }
export class ArgumentCountMismatch implements SemanticalError { }
export class NonVoidFunctionReturnsNothing implements SemanticalError { }
export class RecursiveCall implements SemanticalError { }
```

Listagem 3: Erros semânticos representados por classes

5 Testes

Escreveu-se um programa a partir do *AnaliseDeTexto* fornecido pelo professor. Ele foi expandido para conter exemplos de transgressão a todas as regras semânticas especificadas.

No total, o programa contém 60 casos de transgressão devidamente identificados e localizados no código-fonte. Por brevidade, disponibiliza-se o link https://aptlimepublishers.megatron0000.repl.co, em que se pode examinar o programa-fonte e o resultado da análise semântica, bem como visualizar a tabela de símbolos do programa.

Observa-que a listagem da tabela de símbolos (contida no link) é propositalmente “vaga”: a tabela, como já dito, não armazena informações que dizem respeito a regras semânticas (como contagem de referências a um símbolo), porque essas informações são calculadas pelas classes que verificam a obediência às regras. Nem armazena o tipo de cada símbolo, pois essa informação é obtida da árvore de sintaxe abstrata.

Referências

[1] Mokarzel, F. C. **Linguagem COMP-ITA 2019**. Especificação formal de uma linguagem de programação utilizada na disciplina CES-41 do Instituto Tecnológico de Aeronáutica, 2019.

[2] Mokarzel, F. C. **4º Laboratório de CES-41/2019**. Roteiro do laboratório 4 da disciplina CES-41 do Instituto Tecnológico de Aeronáutica, 2019.

[3] Radvan, T. **Moo**. Analisador léxico escrito em *javascript*. Repositório em: <https://github.com/no-context/moo>. Acesso em 04/03/2019.

[4] Hardmath123. **Nearley**. Analisador sintático em *javascript*. Repositório em: <https://github.com/vihanb/nearley>. Acesso em 09/04/2019.