Sumário

Estrutura

O projeto segue a estrutura de **diretórios** descrita abaixo:

- `fontes`: arquivos fontes do projeto.
    - `fontes/java`: arquivos fontes do **Java**.
    - `fontes/g`: gramáticas do **Antlr**.
    - `fontes/caneca`: arquivos fontes da nossa linguagem.
- `bibliotecas`: bibliotecas utilizadas no projeto.
- `binarios`: arquivos binários resultantes da compilação dos arquivos fontes.
- `construcao`: arquivos temporários gerados na construção.
- `recursos`: arquivos extras como imagens, documentos, apresentações e outros utilizados no projeto.
- `documentacao`: arquivos de documentação.
- `gerados`: arquivos gerados a partir da execução do **Compilador**.

Vale destacar alguns **arquivos** importantes do projeto:

Sumário ⚓

- `fontes/java/.../Compilador.java`: classe responsável por compilar um arquivo **Caneca** passado como parâmetro.
- `fontes/g/CanecaLexico.g`: gramática contendo a especificação léxica da linguagem.
- `fontes/g/CanecaSintatico.g`: gramática contendo a especificação sintática da linguagem.
- `fontes/g/CanecaArvore.g`: gramática contendo a especificação sintática da linguagem utilizando regras de reescrita para geração da árvore.
- `fontes/java/.../antlr/CanecaLexico.java`: analisador léxico gerado pelo **Antlr** com base no arquivo `fontes/g/CanecaLexico.g`.
- `fontes/java/.../antlr/CanecaSintatico.java`: analisador sintático gerado pelo **Antlr** com base no arquivo `fontes/g/CanecaSintatico.g`.
- `fontes/java/.../antlr/CanecaArvore.java`: analisador sintático com regras de reescrita gerado pelo **Antlr** com base no arquivo `fontes/g/CanecaArvore.g`.

Antlr

No projeto foi utilizado o **Antlr** para gerar os analisadores. O **Antlr** recebe como entrada uma gramática e a partir dela gera os analisadores léxicos e sintáticos. É possível especificar em qual linguagem o **Antlr** irá gerar os analisadores. No nosso caso optamos pela linguagem **Java**. Para utilizar o **Antlr** e gerar um analisador, basta digitar o seguinte comando no terminal:

`java -classpath bibliotecas/jar/atlr.jar org.antlr.Tool fontes/g/CanecaLexico.g -fo fontes/java/br/ufsc/inf/ine5426/caneca/antlr/`.

Ou, de forma genérica:

`java -classpath` **`<bibliotecaDoAntlr>`** `org.antlr.Tool` **`<gramatica>`** `-fo` **`<destinoDoAnalisadorGerado>`**.

Como utilizaremos analisadores geradores na linguagem **Java** pelo **Antlr**, é importante conhecer as principais classes da API:

- `CharStream`: Abstração de um fluxo de caracteres que é utilizado pelo `Lexer`.
- `ANTLRFileStream`: Extensão de `CharStream` que representa um fluxo de caracteres proveniente de um arquivo.

- `TokenStream`: Abstração de um fluxo de símbolos que é utilizado pelo `Parser`.
- `CommonTokenStream`: Extensão de `TokenStream` que representa um fluxo de símbolos proveniente de um analisador léxico.
- `TreeNodeStream`: Abstração de um fluxo de nodos que é utilizado pelo `TreeParser`.
- `CommonTreeNodeStream`: Extensão de `TreeNodeStream` que representa um fluxo de nodos proveniente de um analisador sintático.
- `Lexer`: Abstração de um analisador léxico que é estendido pelo analisador léxico gerado pelo **Antlr**.
- `Parser`: Abstração de um analisador sintático que é estendido pelo analisador sintático gerado pelo **Antlr**.
- `TreeParser`: Abstração de um analisador sintático que permite a reescrita de regras e é extendido pelo analisador sintático árvore gerado pelo **Antlr**.
- `BaseRecognizer`: Abstração de um analisador básico que é estendido pelo `Lexer`, `Parser` e `TreeParser`.

Construção

O projeto possui em sua pasta raiz o arquivo `construir.sh`. Esse arquivo é um *script* criado para realizar a compilação do código fonte e a geração dos analisadores do **Antlr**. Para executá-lo basta digitar no terminal `$ ./construir.sh` (em ambiente **Linux**).

Após a execução do arquivo `construir.sh` os arquivos fontes são compilados e os analisadores do **Antlr** são gerados. Para ver detalhes sobre como os analisadores do **Antlr** são gerados você pode ver a seção Antlr ⚓ deste documento.

Execução

Para a execução do projeto foi criado o arquivo `executar.sh` na pasta raiz do projeto. O *script* executa o **Compilador** que por sua vez recebe como parâmetros arquivos fontes a serem compilados. Assim, você poderá utilizar o *script* através do seguinte comando:

`$ ./executar.sh` **<classeCanecaASerCompilada> <opcaoDoCompilador>**.

O *script* `executar.sh` irá realizar a construção do projeto e posteriormente irá executar o **Compilador** através do seguinte comando:

```
$ java -classpath binarios/class:bibliotecas/jar/antlr
br.ufsc.inf.ine5426.caneca.Compilador <classeCanecaASerCompilada>
```

<opcaoDoCompilador> .

Nesse primeiro momento o **Compilador** realiza apenas a análise léxica e imprime os símbolos e lexemas na tela. ⊖ A medida que o projeto for avançando as tarefas de análise sintática, análise semântica e geração de código serão adicionadas ao **Compilador**. ⊖

Foram adicionadas opções ao **Compilador** ⊕. Agora, ao executar o **Compilador** é necessário informar a ação desejada. ⊕ As opções disponíveis são as que seguem: ⊕

- `lexica`: Realiza a análise léxica do arquivo fonte e imprime na tela os erros léxicos, caso existam.
- `sintatica`: Realiza a análise léxica e sintática do arquivo fonte e imprime na tela os erros léxicos e sintáticos, caso existam.
- `simbolos`: Realiza a análise léxica do arquivo fonte, gera o arquivo `gerados/simbolos.html` que contém os símbolos reconhecidos na análise e imprime na tela os erros léxicos, caso existam.
- `arvore`: Realiza a análise léxica e sintática do arquivo fonte e gera o arquivo `gerados/arvore.html` que contém a árvore sintática resultante da análise. Caso haja algum erro de compilação, então a árvore não é gerada.
- `semantica`: Análise semântica com exibição dos erros no console.
- `geracaoDeCodigo`: Geração de código de máquina.
- `execucao`: Geração de código de máquina e execução do código gerado.

Análise léxica

A gramática léxica da nossa linguagem pode ser encontrada em `fontes/g/CanecaLexico.g`. A partir da gramática léxica foi gerado através do **Antlr** o analisador léxico que pode ser encontrado em `fontes/java/.../antlr/CanecaLexico.java`.

Palavras-chave, operadores e símbolos auxiliares

Para a criação da gramática léxica primeiramente definimos os símbolos que são **palavras-chave** da nossa linguagem como por exemplo, `classe`, `metodo`, `enquanto`, `importe` e outros. Também foram definidos os **operadores aritméticos** como soma `+` e multiplicação `*`, e os **operadores lógicos** como negação `~`, e lógico `&&` e maior igual `>=`. Definimos também outros tipos de **símbolos auxiliares** para a linguagem como os parenteses `(` e `)`, os colchetes `[` e `]`, símbolo de atribuição `=` e

outros. Também foi especificados o símbolo ponto `.` para o uso de métodos e atributos de **objetos** e o símbolo dois pontos `:` para uso de métodos e atributos de **classe**.

Identificadores e identificadores de pacote

Os *tokens* descritos anteriormente são construções fixas particulares da nossa linguagem. O nosso segundo passo foi definir os *tokens* variáveis como **identificadores** que são iniciados por uma letra e são seguidos por zero ou mais letras, números ou sublinhados. Os **identificadores** são usados pelo programador da linguagem para definir nomes de: classes, variáveis, métodos e atributos.

Temos também os **identificadores de pacote** que são prefixados com o arroba `@` e são formados por um ou mais **identificadores** separados por ponto `.`. Eles são usados para declarar o pacote que uma dada classe pertence e para explicitar o pacote de uma classe utilizada. A utilização de pacotes na linguagem é útil caso você utilize no seu código duas classes diferentes, porém com o mesmo nome. Por exemplo, suponha que você utilize no seu código uma classe chamada **Lista** que faz parte do pacote de estruturas de dados e utilize também uma classe **Lista** que faz parte do pacote de interface gráfica. Sem explicitar o pacote não seria possível definir qual classe **Lista** está sendo utilizada em um determinado momento do código.

Valores, constantes e literais

O próximo passo foi realizar a definição dos valores primitivos que poderão ser utilizados em nossa linguagem. Definimos o **valor** `nulo` e os **valores booleanos** `verdadeiro` e `falso`. Definimos também as **constantes inteiras** que são uma sequência de números com o prefixo opcional sinal negativo `-` e as **constantes reais** que são sequências de números com o prefixo opcional sinal negativo `-` seguidos pelo separador decimal obrigatório ponto `.` e seguido por uma sequência de números.

O **literal caractere** e o **literal texto** também foram definidos nessa terceira etapa. O **literal caractere** é composto por aspas simples `'` seguido por um **caractere** ou um **caractere de escape** e finalizado por aspas simples `'`. Já o **literal texto** é iniciado por aspas duplas `"`, seguido por uma sequência de **caracteres** e **caracteres de escape** e finalizado por aspas duplas `"`.

É interessante mencionar que nessa parte utilizamos um recurso do **Antlr** que é o `fragment`. O `fragment` permite que seja definido um *token* que nunca será enviado ao analisador léxico, pois esse *token* será utilizado na construção de outros *tokens*. Por exemplo, os *tokens* **literal caractere** e **literal texto** possuem em sua regra de sintaxe a utilização do *token* **caractere**. Isso é possível através do uso do `fragment`, pois com

ele o **Antlr** saberá que ao encontrar um **caractere** ele deverá encaixá-lo em um **literal caractere** ou em um **literal texto**, mas não deverá nunca transformá-lo em um *token* próprio.

Uma importante observação a respeito do **literal caractere** e do **literal texto** é que neles podem ser usados caracteres especiais como o `\n` que representa quebra de linha e o `\t` que representa tabulação. Em contrapartida não é possível utilizar esses caracteres escapados como componentes dos literais. Ou seja, ao iniciar um **literal texto** é possível definir, por exemplo, `\n`, mas não é possível usar a quebra de linha propriamente dita. A utilização desses caracteres especiais traz um problema que é a impossibilidade de utilizar o caractere barra invertida `\` já que ele é utilizado como prefixo de um caractere especial. Por isso, é preciso ter uma sequência de escape para o barra invertida, que no casso é `\\`.

Outros caracteres que precisam ser "escapados" em um literal são as aspas. Isso é necessário, pois as aspas são delimitadores dos literais e uma forma de fazer com que o analisador saiba que uma aspas é apenas um caractere que compõe um texto e não um delimitador é escapando as aspas: `\'` e `\"`. Os caracteres que podem ser escapados com barra invertida `\` são `n`, `r`, `t`, `f`, `\`, `'`, `"`, `a`, `b`, `e` e `v`.

Espaços em branco e comentários

A última etapa na definição da nossa gramática léxica foram os **comentários** e **espaços em branco**. Os **comentários** podem ser em bloco ou em linha. Um **comentário em linha** será prefixado pelo símbolo interrogação `?` e será composto por qualquer coisa que não seja uma quebra de linha. Já um **comentário em bloco** é iniciado por sustenido e interrogação `#?` e finalizado por interrogação e sustenido `?#`. Para o **comentário em bloco** utilizamos uma opção do **Antlr** que é `(options {greedy=false;} ...)`. O que essa opção faz é dizer para o analisador léxico não ser "guloso" e formar o *token* **comentário em bloco** assim que possível. Exemplificando, a opção `(options {greedy=false;} ...)` faz com que o código `#? identificadorA ?# identificadorB #? identificadoC ?#` possua três *tokens*: o primeiro e o último um **comentário em bloco** e o segundo um **identificador**. Se essa opção não tivesse sido utilizada o analisador iria reconhecer apenas um *token* que seria o **comentário em bloco**.

A diretiva `{$channel=HIDDEN;}` foi utilizada nos comentários e o que ela faz é adicionar o *token* a um "canal não visível" de forma que por padrão o analisador sintático não enxergue esse *token*.

Os espaços em branco são compostos por **espaços**, **tabulações** e **quebras de linha**.

Para esse tipo de *token* foi utilizada a diretiva `{skip();}` do **Antlr**. Essa diretiva faz com que o *token* seja reconhecido e ignorado. Repare que o comportamento do `{skip();}` é diferente do `{$channel=HIDDEN;}`, pois no caso deste último, o *token*, embora escondido por padrão, ainda é enviado ao analisador sintático que pode solicitar o *token* de forma explicita. No caso do `{skip();}` o *token* sequer é enviado ao analisador léxico.

Análise sintática

Logo na primeira semana do desenvolvimento deste trabalho a gramática sintática foi definida através da notação EBNF. A definição inicial da gramática foi relativamente fácil, porém em geral, os geradores de analisadores sintáticos possuem algumas restrições quanto a gramática de entrada e isso fez com que nossa gramática inicial precisasse ser modificada. Antes de comentar os problemas que encontramos ao realizar as modificações na gramática convém explicar alguns mecanismos do **Antlr**.

O **Antlr** possui mais de um modo para gerar analisadores sintáticos:

1. Analisador descendente recursivo com **backtracking**.
2. Analisador descendente recursivo preditivo **LL(k)** com o k definido pelo usuário.
3. Analisador descendente recursivo preditivo **LL(*)** onde não é restringido o valor de k, mas as não fatorações devem ser uma gramática regular.

A primeira ou a terceira opção seriam suficientes para gerar um analisador sintático para o nossa primeira gramática que foi definida logo no começo deste trabalho sem que houvesse necessidade de maiores modificações. Porém, não queríamos usar nem a opção do **backtracking** nem a opção **LL(*)** pois, estas duas são muito onerosas (principalmente o **backtracking**) em termos de processamento. Devido a isso, o nosso objetivo foi gerar uma gramática **LL(k)** tendo em vista alcançar o menor k possível.

A gramática sintática que críamos pode ser encontrada em `fontes/g/CanecaSintatico.g` e o analisador gerado pelo **Atnlr** a partir desta gramática se encontra em `fontes/java/.../antlr/CanecaSintatico.java`.

Gramáticas LL(k)

Uma gramática LL(k) deve seguir algumas restrições conforme abaixo:

- Não possuir recursão à esquerda.
- Possuir no máximo k-1 *tokens* não fatorados.
- Não ser ambígua.

Expressões

A primeira mudança necessária em nossa gramática inicial foi a remoção da recursão à esquerda. Mais precisamente, as **expressões** foram definidas contendo recursão à esquerda: `expressao : expressao (SOMA | SUBTRACAO | ... | MENOR) expressao ;`. A remoção desse tipo de recursão à esquerda foi facilmente realizada através da definição de uma **expressão primária** contendo apenas símbolos terminais: `expressaoPrimaria : VALOR_BOOLEANO | VALOR_NULO | ... | LITERAL_TEXTO ;`. Dessa forma, a nossa **expressão** passou a ser `expressao : expressaoPrimaria (SOMA | SUBTRACAO | ... | MENOR) expressao ;`.

Também fizemos algumas modificações para definir a ordem de precedência dos operadores. Para isso, definimos uma cadeia de **expressões** de forma que aquelas de menor precedência possuem em sua composição a **expressão** seguinte de maior precedência, por exemplo:

1. `expressaoAditiva : expressaoMultiplicativa ((SOMA | SUBTRACAO) expressaoMultiplicativa)*`
2. `expressaoMultiplicativa : expressaoUnaria ((MULTIPLICACAO | DIVISAO | RESTO_DA_DIVISAO) expressaoUnaria)*`

E assim por diante, até chegar na **expressão primária**. A ordem completa de precedência (do menos precedente para o mais precedente) é a que segue:

1. atribuição: `=`.
2. ou lógico: `||`.
3. e lógico: `&&`.
4. comparação lógica: `==`, `!=`, `>`, `>=`, `<`, `<=`.
5. aditiva: `+`, `-`.
6. multiplicativa: `*`, `/`, `%`.
7. unária: `~`, `-`.
8. primária: `(expressao)`, `VALOR_BOOLEANO`, `VALOR_NULO`, `CONSTANTE_INTEIRA`, `CONSTANTE_REAL`, `LITERAL_CARACTERE`, `LITERAL_TEXTO`, `selecao`, `chamada`.

É importante dizer que podem existir algumas inconsistências semânticas, como por exemplo: `3 + 2 = esse.metodo();`. Para esses e outros casos de inconsistência ficará a cargo do analisador semântico realizar a verificação. Nesse casso especificamente, o analisador semântico precisará verificar se o lado esquerdo de uma **expressão de atribuição** é uma variável.

Não fatoração entre declarações e expressões

Com as modificações realizadas na nossa gramática inicial conseguimos transformá-la em uma **LL(2)**. Tentamos reduzi-la para **LL(1)** porém, não conseguimos realizar a fatoração entre **declarações** e **expressões**. A não fatoração está no fato de quem ambos podem começar com um `IDENTIFICADOR`. No caso de uma **declaração**, o `IDENTIFICADOR` é o tipo da variável e no casso de **expressões** ele pode ser o nome de uma variável ou de um método.

Como não conseguimos fatorar o `IDENTIFICADOR` das **declarações** e **expressões** analisamos uma outra solução que consiste em adicionar o prefixo `declare` para as **declarações**. Com isso, removeríamos a não fatoração e a gramática se tornaria **LL(1)**. Entretanto, chegamos a conclusão de que não valeria a pena pagar o preço do programador ter que utilizar um prefixo toda vez que for declarar uma variável em troca de uma análise levemente mais rápida com uma gramática **LL(1)**.

Uma opção interessante fornecida pelo **Antlr** é especificar valores diferentes do k para diferentes locais da gramática. No nosso caso, utilizamos como k global o valor 1, porém dentro da produção `instrucao`, na subprodução `expressao TERMINADOR | declaracao TERMINADOR`, modificamos o valor de k para 2. Para modificar localmente o valor de k utilizamos a diretiva `(options {k = 2;}: ...)`.

Tratamento de erros

O **Antlr** já possui um tratamento de erros padrão. Ao encontrar um símbolo inválido o **Antlr** elimina o símbolo em questão e passa a eliminar todos os símbolos seguintes até encontrar um *follow* do símbolo inválido. Além disso, o **Antlr** trata de forma especial dois casos: `UnwantedTokenException` e `MissingTokenException`. Ambas exceções estendem `MismatchedTokenException`.

Para os casos de `UnwantedTokenException` o **Antlr** utiliza a estratégia de tratamento por remoção. Ao encontrar um símbolo não desejado é verificado se o próximo símbolo era o que estava sendo procurado. Caso seja, então o símbolo indesejado é eliminado e a análise continua. Por exemplo, ao encontrar o código `10 * (2 + 4));` verifica-se que há um parêntese sobrando. O **Antlr** verifica que o símbolo após o indesejado é o símbolo esperado e com isso, elimina o parêntese extra.

No caso de `MissingTokenException` será utilizada a estratégia de tratamento por inserção, onde o símbolo faltante é inserido. No exemplo `10 * (2 + 4;` o **Antlr**

detecta que o símbolo esperado era o parêntese e insere o símbolo para que a análise prossiga.

O **Antlr** utiliza os métodos abaixo para realizar o tratamento de erros. Assim, sobrescrevemos esses métodos no `fontes/g/CanecaSintatico.g` para modificar o tratamento de erros padrão que é dado pelo **Antlr**. Para tanto, é utilizada a diretiva `@members { ... }` que permite modificar elementos da classe que será gerada pelo **Antlr** e desta forma sobrescrever os métodos de tratamento.

- `void recover(IntStream entrada, RecognitionException erro)`
- `Object recoverFromMismatchedToken(IntStream entrada, int tipoDoSimbolo, BitSet conjuntoDeFollows)`
- `Object recoverFromMismatchedSet(IntStream entrada, RecognitionException erro, BitSet conjuntoDeFollows)`

Optamos por utilizar apenas o tratamento de erros que elimina os símbolos até encontrar um *follow* do símbolo inválido. Por isso, desabilitamos os casos especiais de tratamento do **Antlr** através da sobrescrita dos dois últimos métodos acima. Na sobrescrita apenas lançamos uma exceção. Para capturar as exceções que são lançadas no processo de reconhecimento é utilizada a diretiva `@rulecatch { ... }` do **Antlr**. A exceção lançada nos dois últimos métodos de recuperação será capturada e duas ações serão tomadas: reportar o erro e chamar o método `recover` para que seja dado o tratamento de erros padrão.

Para os erros léxicos o tratamento de erros consiste apenas em consumir o símbolo inválido e reportar o erro. O método para tratamento de erros léxicos é o `void recover(RecognitionException erro)`.

Também realizamos modificações para melhorar as mensagens de erros dadas pelo **Antlr**. Para isso, bastou sobrescrever o método `String getErrorMessage(RecognitionException erro, String[])`.

Árvore sintática

Para a geração da árvore sintática criamos um novo arquivo, o `fontes/g/CanecaArvore.g`. Essa gramática possui as mesmas regras sintáticas do `fontes/g/CanecaSintatico.g`, porém possui a diferença de utilizar o modo de reescrita do **Antlr**. O modo de reescrita permite manipular os símbolos já reconhecidos e realizar ações como: modificar, reordenar, omitir, duplicar entre outras. É importante dizer

que o modo de reescrita não altera a linguagem ou a gramática reconhecida, ela altera apenas a saída que no nosso caso vai ser uma árvore.

Para definir que a saída gerada a partir de um reconhecimento será uma árvore, colocamos dentro da diretiva `options { ... }` do **Antlr**, a opção `output = AST;`. A menos que se especificado algo contrário, a árvore gerada pelo **Antlr** será uma *flat tree*. Ou seja, será um árvore que terá um nodo pai e todos os demais nodos estarão no mesmo nível (como se fosse uma lista). Para criar uma árvore mais adequada é preciso especificar quais são os nodos raiz das sub-regras. O **Antlr** permite fazer isso através do modo de reescrita com a utilização do operador `^`.

Por exemplo, na sub-regra `listaDeParametros : PARENTESE_ESQUERDO (expressao (SEPARADOR expressao)*)? PARENTESE_DIREITO -> ^(PARAMETROS_ (expressao)*) ;` utilizamos o modo de reescrita para manipular a árvore sintática gerada. É a partir dos símbolos `->` que é iniciado o modo de reescrita. O operador `^` indica que o primeiro item dos que seguem será o nodo raiz da sub-regra. Repare também que na reescrita da sub-regra foram omitidos os parenteses e separadores. Isso foi feito, pois não existe a necessidade de incluir estes símbolos na árvore sintática já que eles apenas se fazem necessário para dar mais clareza a linguagem.

Análise semântica

Através da árvore sintática gerada na etapa anterior, partimos para a etapa de análise semântica. O **Atnlr** permite percorrer pela árvore gerada e realizar algumas operações com os nodos, como eliminar nodos desnecessários, duplucar alguns, alterar a posição e assim por diante. Essas operações são importantes, pois permitem que seja feita a decoração da árvore.

No **Antlr** é possível caminhar pela árvore sintática através de duas regras que são especificadas na própria gramática. Nas regras `bottomup` e `topdown` (pré-definidas pelo **Antlr**) você coloca outras regras que serão percorridas na decoração da árvore. Essas outras regras por sua vez poderão conter código **Java** e dessa forma é possível criar uma estrutura de dados própria. A diferença entre `bottomup` e `topdown` é que na primeira a árvore é percorrida de baixo para cima e na segunda de cima para baixo.

O que fizemos então, foi devidir a análise samântica em várias etapas e caminhar pela árvore em cada uma dessas etapas. A primeira etapa é a de **definição** e nela montamos a nossa tabela de símbolos obtendo as classes, variáveis, atributos, métodos, etc. Na etapa da **resolução** são resolvidas as referências a variávies, classes e métodos. A

terceira esta consiste da **verificação estática das expressões** e por fim, na quarta etapa, são feitas verificações extras como **checagem de parâmetros** e **chamadas**.

Na nossa linguagem, pelo menos duas passadas (etapas são necessárias), pois existe a possíbilidade de realizar referências avançadas. No entanto, decidimos relaizar o processo todo em quatro etapas, para facilitar a escrita e manutenção do código.

Na parte da definição, utilizamos o conceito de tabela de símbolos e de escopo. Definimos que a tabela de símbolos será simplesmente um escopo global. Os escopos conterão outros escopos e também conterão a definição de variáveis, métodos, classes, etc. Classes, métodos, blocos, construtores e destrutores também são escopos. A tabela de símbolos possui em seu escopo apenas classes. As classes possuem em seu escopo atributos e métodos. Enquanto que os métodos possuem em seu escopo os argumentos e os blocos de instruções. Os blocos de instruções possuem em seu escopo variáveis e outros blocos de instruções. O processo de resolução consiste em verificar se determinada classe, atributo, método ou variável se encontra nos escopos superiores.

Para a verificaçõa estática dos tipos foi utilizado um conceito semelhante ao conceito dos escopos. No nosso caso, cada expressão possui um ou dois operandos sendo que estes podem ser elementos da linguagem ou até mesmo outras expressões. Cada tipo de expressão é reponsável por verificar se os seus operandos respeitam os símbolos exigidos.

Após a realização das análises anteriores, a verificação de parâmetros de chamadas de métodos e de instaciações se tornou mais simples. Cada expressão sabe determinar o seu tipo. A lista de parâmetros é formada por expressões. Com isso, basta verificar se os tipos dos parâmetros combinam com a assinatura do método ou do construtor que se deseja invocar.

Geração de código

O primeiro passo para a geração de código foi a criação de uma estrutura de dados própria, para representar os elementos da nossa linguagem. Fizemos isso para não ficar acoplado as estruturas de dados do **Antlr**. Assim, através da arvore sintática, passamos por cada nodo desta árvore e fomos adicionando os nodos à nossa estrutura da dados. Por exemplo, ao percorrer um `atributo`, adicionamos esse `atributo` em um elemento pai, que é a `classe`.

Tendo criado uma classe **Java** para cada elemento da nossa linguagem, adicionamos um método a esses elementos, e esse método é `gerarCodigo`. Ou seja, cada elemento sintático da nossa linguagem sabe gerar o próprio código.

O próximo passo foi decidir a linguagem alvo para o qual a nossa linguagem será traduzida. Decidimos então criar a nossa própria linguagem de máquina e, desta forma, a nosssa própria máquina virtual para interpretar a linguagem de máquina.

As instruções disponíveis para a `MaquinaCaneca` são as que seguem:

- `abrirContexto`: abre o contexto para blocos de instruções. Isso permite criar uma hierarquia de blocos aninhados onde é possível ter símbolos definidos com o mesmo nome em vários níveis. O contexto pai do contexto aberto será aquele que estiver no topo da `pilhaDeContextos`.
- `abrirContextoDeProcedimento`: abre um contexto para um procedimento que será executado. O pai do novo contexto será o objeto `esse`.
- `atribuir`: obtém o valor e a referência na `pilhaDeDados` e atualiza a referencia no contexto que está no topo da `pilhaDeContextos`.
- `chamar {nomeDoProcedimento} {pontoDeRetorno}`: adiciona o ponto de retorno na `pilhaDeExecucao` e busca no contexto que está no topo da `pilhaDeContextos` o ponto de entrada do procedimento desejado.
- `definirSimbolo {nome}`: define um símbolo no contexto do topo da `pilhaDeContextos`. O valor da definição será buscado na `pilhaDeDados`.
- `depurar`: imprime na tela informações de depuração.
- `desempilhar`: desempilha um valor da `pilhaDeDados`.
- `desviar {pontoDeDesvio}`: desvio incondicional que manipula o `contadorDePrograma` para que a próxima instrução seja buscada no ponto de desvio fornecido.
- `desviarSeFalso {pontoDeDesvio}`: faz o mesmo que o desviar, porém busca na `pilhaDeDados` o próximo valor. Se este valor for falso, então o desvio será realizado.
- `desviarSeFalso {pontoDeDesvio}`: faz o mesmo que o `desviarSeFalso`, porém só desvia se o próximo valor na `pilhaDeDados` for verdadeiro.
- `dividirI`, `dividirR`, `somarI`, `somarR`, `multiplicarI`, `multiplicarR`, `subtrairI`, `subtrairR`: operações aritméticas de inteiros e números reais obtidos na `pilhaDeDados`, onde o resultado é colocado na `pilhaDeDados`.
- `imprimir`: imprime na tela o valor no topo da `pilhaDeDados`.
- `duplicar`: duplica o valor disponível no topo da `pilhaDeDados`.
- `extrairContexto`: extraí o contexto da referência no tipo da `pilhaDeDados` e coloca no topo da `pilhaDeContextos`.
- `fecharContexto`: fecha o contexto aberto, removedo-o da `pilhaDeContextos`.

- `igual`, `diferente`, `maior`, `menor`, `maiorQue`, `menorQue`: operador relacionais que buscam dois operandos na pilhaDeDados e deixam o resultado na mesma `pilhaDeDados`.
- `instanciar {nomeDaClasse}`: instancia um objeto da classe desejada criando um contexto para o objeto e colocando-o na `pilhaDeContextos`.
- `resolverSimbolo {nome}`: busca o símbolo no contexto do topo da `pilhaDeContextos` e adiciona o valor do símbolo na `pilhaDeDados`.
- `retornar`: busca ponto de retorno na `pilhaDeExecucao` e altera o `contadorDePrograma` com o endereço obtido.
- `sair`: encerra e execução da `MaquinaCaneca`.

Execução

A execução se dá através da `MaquinaCaneca` que foi implementada por nós. A máquina segue a arquitetura baseada em pilha. Nessa arquitetura cada operação deixa no topo da pilha o resultado. Por exemplo, considerando a seguinte operação: `3 + 4`. O `3` é um operação que coloca o próprio 3 no topo da pilha. O mesmo acontecerá com o número 4. Por fim, o `+` é uma operação que consumirá dois operandos da pilha e deixará no topo dela o resultado, que neste caso é 7. O código de máquina gerado para a operação anterior será:

- `empurar 3`
- `empurar 4`
- `somar`

A `MaquinaCaneca` possui alguns elementos básicos, sendo eles:

- **pilhaDeDados**: Onde são colocados os dados das expressões. Os operandos de uma adição, por exemplo, são colocados na **pilhaDeDados** e posteriormente são consumidos pelo comando `somar`.
- **pilhaDeContextos**: São empilhados os diversos contextos, incluindo contextos de objetos e de métodos. Cada vez que um método ou objeto é chamado, o seu contexto é empilhado. Os contextos são responsáveis por guardar símbolos (atributos e variáveis) e até mesmo outros contextos. Uma classe, por exemplo, guarda os pontos de entrada dos seus métodos. Os contextos também possuem uma hierarquia, da mesma forma que existe nos escopos, que foram utilizados na etapa da análise semântica.
- **pilhaDeExecucao**: Onde são empilhados os pontos de retorno dos procedimentos executados. Cada vez que um procedimento é chamado, é

empilhado um ponto de retorno. Quando houver o retorno do procedimento, o ponto de retorno será desempilhado e a execução passará a apontar para a instrução imediatamente após a chamada.

- **areaDeDados**: É nessa área que serão colocados os atributos, objetos e outros símbolos. A **areaDeDados** nada mais é do que um contexto global. Dentro dela existe uma hierarquia de outros contextos onde cada um é responsável pelos seus símbolos.
- **areaDeCodigo**: Local onde é carregado todo o código do programa. A **areaDeCodigo** é apenas um vetor onde cada posição contém uma instrução de máquina.

A `MaquinaCaneca` também possui um `contadorDePrograma` que aponta para a próxima instrução de máquina a ser executada. A execução é tipicamente sequencial, de tal forma que as instruções são executadas na mesma ordem que foram definidas na `areaDeCodigos`. Entretanto, existem instruções de máquina que manipulam o `contadorDePrograma` e dessa forma realizam o controle de fluxo de execução.

O *entry point* de uma programa Caneca é o construtor de uma classe passada como parâmetro no momento da compilação. Na Caneca não existe o método `main`, por isso `main` o ponto de entrada é o construtor de uma classe. A definição do ponto de entrada estará no inicio da `areaDeCodigos`. A execução de um programa Caneca irá durar enquanto o construtor do ponto de entrada estiver ativo.

Após terminar a execução do construtor do ponto de entrada, será desempilhado o ponto de retorno da `pilhaDeExecucao`. A execução será então direcionada para esse ponto de retorno que aponta para uma instrução de máquina responsável por encerrar a execução da `MaquinaCaneca`.

O coração da MaquinaCaneca consiste de um simples método:

- `public void executar() {`
- `Integer fimDoCodigo = areaDeCodigo.size();`
- `pilhaDeContextos.push(areaDeDados);`
- `while (contadorDePrograma < fimDoCodigo) {`
- `Codigo codigo = areaDeCodigo.get(contadorDePrograma);`
- `contadorDePrograma++;`
- `codigo.executar(this);`
- `}`
- `}`

Todo o resto é realizado por implementações da classe Codigo. O código, quando for executado terá acesso à: `areaDeDados`, `areaDeCodigo`, `pilhaDeDados`,

`pilhaDeExecucao`, `pilhaDeContextos` e `contadorDePrograma`.