Linguagem para Manipulação de Tabelas - Relatório

Licenciatura em Engenharia Informática
41469 - Compiladores

Camila Uachave - 90711
João Nogueira - 89262
José Frias - 89206
Marta Ferreira - 88830
Pedro Miguel – 89069
Tiago Melo - 89005

Docentes:
André Zúquete
Artur Pereira
Miguel Oliveira e Silva

Índice

Introdução

Um compilador consiste num programa que, essencialmente, "traduz" uma linguagem noutra (linguagem-alvo), geralmente mais baixo-nível. Para isto acontecer, é necessário que o programa a traduzir obedeça a certas regras, sintáticas e semânticas, de forma a que este seja válido e de possível compilação. Por outras palavras, compiladores inferem a correção de dadas sequências de símbolos pertencentes ao alfabeto da linguagem a compilar e, caso sejam válidas, procedem à "supracitada" tradução.

Posto isto, projetámos um compilador para uma linguagem de manipulação de tabelas, que denominámos de UTL (Unified Table Language). Estabelecemos como funcionalidades cruciais a definição de variáveis do tipo tabela, exportar tabelas para ficheiros csv e operações entre tabelas (como join ou seleções). É de salientar que, ao longo do desenvolvimento do projeto, nos baseámos em linguagens já existentes que permitem operações semelhantes e que se apoiam nas mesmas premissas, como SQL e MATLAB.

Gramática

A análise sintática da nossa linguagem é feita usando uma gramática desenvolvida em ANTLR4. É de salientar que foi a partir desta ferramenta que UTL ganhou forma, sendo útil tanto no desenvolvimento da gramática como na geração dos ficheiros (visitors) que mais tarde foram usados para efetuar a análise semântica e a geração de código, como iremos ver posteriormente.

Aquando do desenvolvimento da gramática que representaria UTL, considerámos que o nosso programa seria composto por vários "r"'s, sendo este o nome da nossa mainRule.

Um "r", por sua vez, poderá ser qualquer sequência de operações `declaration`; `operation`; `assignment`; `print`. Estas correspondem, respetivamente, a declarações, operações, atribuições e prints (imprimir algo no terminal).

Declarações

Considerámos que uma declaração consistia na primeira referência a uma variável, sendo por isso obrigatório referir o tipo da mesma nesta instrução. Para além disso, possibilitámos a declaração de Tabelas, Colunas, Itens. Uma Tabela pode ser declarada através da enumeração de uma série de colunas que a constituem (que podem ser "criadas" dentro da declaração da tabela), ou através da junção de duas outras tabelas.

As Colunas podem ser declaradas através do acesso a Tabelas. Podemos declarar uma nova coluna referenciando uma coluna pertencente a uma Tabela já existente (resultando em que a nova coluna se torne uma cópia da coluna já existente) ou através da utilização de uma gama, restringindo os conteúdos (Itens) que irão ser copiados para a nova coluna. Por outro lado, a coluna também pode ser declarada por extenso, se o programador declarar os itens pertencentes à nova coluna um a um ou buscando uma variável de coluna já existente. Este é a regra usada quando um Tabela é definida por extenso.

Por fim, os Itens podem ser declarados através do acesso a uma Tabela (de forma semelhante à já vista nas Colunas), diretamente (como qualquer declaração de variáveis em Java: Item x = "value") ou por cópia de variáveis.

Operações

Como já foi mencionado, incluímos em UTL a possibilidade de efetuar operações sobre elementos já criados. Com uma sintaxe semelhante à do Python, funcionam como se fossem funções inerentes à linguagem. Incluímos funções que

eliminam, adicionam ou alteram elementos já existentes. Para além disso, foi aqui incluída a supracitada funcionalidade de exportar uma Tabela para um ficheiro csv.

As funções de eliminação são bastante *straightforward*: consistem essencialmente na referência ao elemento com que pretendemos interagir (através do nome da variável) e invocamos a função `del` sobre o mesmo. É de salientar que só consideramos relevante a eliminação de elementos já pertencentes a Tabelas, daí que qualquer regra pressuponha uma referência a uma variável de Tabela.

Análogamente, as funções de adição de elementos partem de uma referência a uma variável de Tabela. É depois possível adicionar qualquer número (maior que zero) de Colunas ou Itens, sendo obrigatória a referência à Coluna onde deverão ser adicionados os mesmos *in the latter*.

A alteração de elementos é feita de forma semelhante à adição, havendo ligeiras mudanças em palavras chave (`put -> alter`) e no número de elementos nos quais a operação pode ser efetuada (limitado a um).

Por fim, a exportação de tabelas é iniciada novamente com uma referência à tabela que se pretende exportar, e posteriormente é invocada a função `.csv(arg1)`, onde `arg1` é o nome do ficheiro para o qual a tabela irá ser exportada.

Atribuições e Prints

Atribuições permitem associar valores a variáveis que já foram declaradas, o que contribui para a flexibilidade e versatilidade de UTL. Similares às atribuições já definidas em linguagens existentes, `assignments` funcionam, de certa forma, como declarações em que o tipo da variável é omitido.

Por fim, a impressão de variáveis no terminal é feita de forma muito semelhante à do Python, sendo que só é possível a impressão de Tabelas, referenciadas através de uma variável.

Tokens Terminais e Notas

O conjunto de Tokens Terminais consistia essencialmente em: STRINGS delimitadas por aspas, IDs que entram na definição e declaração de variáveis sendo que o primeiro carater nao pode ser um numero nem um *underscore*, INTs (utilizados quando é preciso o uso de indexação, comentários *a la* Bash (com o uso de "#" e o *non-greedy operator*) e a definição de *whitespace,* consistindo em espaços, tabulação e mudanças de linha. Por fim, foi criada uma regra ERROR, permitindo e certificando que cada token, se não cair em mais nenhuma, cai nesta regra, gerando um ERROR.

Análise Semântica

Como foi mencionado acima, um compilador deverá efetuar uma análise sintática e semântica antes de gerar código. A análise sintática é imposta automaticamente pelo ANTLR4 e a gramática já definida. A análise semântica, por outro lado, poderia ser feita de três formas diferentes: pela injeção de código Java na gramática em ANTLR4, através do uso de listeners e através do uso de visitors. Por questões de versatilidade, simplicidade e clareza, optámos pela última estratégia.

A aproximação com o uso de visitors consiste, essencialmente, numa série de funções que são chamadas quando o analisador "entra" numa regra. O ficheiro base (UTLBaseVisitor.java) foi gerado pelo ANTLR4 e é extendido pelas classes SemanticAnalysis.java (responsável pela análise semântica) e UTLCompiler.java (responsável pela geração de código) que será abordado posteriormente. Herança de classes permite-nos a adoção e edição de todos os métodos existentes no UTLBaseVisitor.java e apenas dar @Override aos métodos que pretendemos. É de salientar o uso de de *subrules* na gramática, que, basicamente, indicam ao ANTLR4 que deverá gerar *callbacks* específicas para as regras respetivas. Isto é bastante útil quando estamos perante várias regras semelhantes e queremos simplificar o processo de distinção entre elas.

Considerámos como violação de regras semânticas a delcaração de variáveis ja declaradas, a referência a variáveis ainda não declaradas e a invocação de métodos não definidos para um dado tipo de objetos (por exemplo, tentar aceder às colunas de uma variável que dizia respeito a uma coluna). A associação variável-valor foi feita recorrendo a uma classe por nós definida, a Tabela de Símbolos.

Tabela de Símbolos

Uma tabela de símbolos contém a informação respetiva a cada variável que é criada ao longo da execução de um dado programa. Para além disso, cada "elemento" da nossa linguagem (Tabela, Coluna, Item, que têm uma classe associada) contém um atributo do tipo Type, que permite inferir o tipo de dados de cada objeto. Type, por sua vez, é uma classe enumerado cujos valores são apenas `table, column, item.` Este atributo provou ser de extrema relevância, especialmente no que toca à verificação da terceira das regras semânticas supracitadas.

```
public class SymTable{

    // public SymTable parent;
    protected HashMap<String, Symbol> table;
```

A Tabela de Símbolos em si (SymTable.java) consiste num HashMap<String, Symbol> onde a chave do HashMap (String) corresponde ao nome da variável que se pretende guardar e o valor (Symbol) é a classe mãe dos elementos da nossa linguagem (Tabela, Coluna, Item) e que conterá o valor associado à variável definida. A classe Symbol.java foi extremamente útil neste cenário, visto que permitiu que o tratamento de diferentes tipos de dados fosse levado de forma indistinta: polimorfismo. É de salientar que é na classe Symbol.java que estão contidos os atributos comuns a todas as subclasses (Type e Nome).

```
public class Symbol {
  protected String name;
  protected Type type;

  public Symbol(String name, Type type){

    //assert name != null && !name.isEmpty();
    assert type != null;

    this.name = name;
    this.type = type;
  }
```

Para além disso, o HashMap foi declarado como `protected`, sendo que o acesso ao mesmo é feito através de funções definidas dentro da classe, nomeadamente: verificar se uma dada variável foi definida (`public boolean exists(String name)`) que irá retornar `True` se o HashMap contiver uma String de valor igual a name e `False` caso contrário; ir buscar o valor associado a uma variável (`public Symbol get(String name))` que pressupõe a existência da variável no HashMap (daí o uso de `assert`; adicionar e eliminar variáveis (`public void add/delete(String name))` onde é pressuposta a inexistência/existência da variável a adicionar/eliminar no HashMap, respetivamente.

SemanticAnalysis

Como já foi mencionado, a classe SemanticAnalysis.java é responsável pela imposição das regras semânticas atrás definidas. Algo a distinguir entre a classe SemanticAnalysis e UTLCompiler é o tipo de dados que é passado pelo Visitor em si. Na classe SemanticAnalysis, originalmente considerámos relevante a circulação de dados tipo Boolean (um retorno do valor `True` significa que a visita à regra não gerou erros, o retorno do valor `False` é feito no caso contrário). Posteriormente, na classe UTLMain, responsável pela execução de todas as partes do programa, seria decidido se se seguiria para a geração de código (caso não houvesse erros semânticos i. e. SemanticAnalysis devolvesse True). Contudo, à medida que fomos desenvolvendo o projeto, optámos pelo aborto da execução do programa. Ou seja, invocamos `System.exit(1)` na presença de um erro, garantindo assim que, se a função main "chegar" à geração de código, não há nenhum erro sintático nem

semântico subjacente (o que não significa que o programa em si não resulte em erro, visto que há diferentes tipos de violações de regras que não podem ser detetadas através de regras sintáticas/semânticas, como por exemplo IndexOutOfBoundsException, NullPointerException, etc).

Para além disso, a classe SemanticAnalysis tem um atributo do tipo SymTable, onde irá instanciar uma tabela de simbolos e manter lá as referências para cada um deles.

Iremos explicar agora como procedemos para a verificação e asserção de cada uma das regras acima mencionadas através do uso da classe SemanticAnalysis.

Declaração de variáveis já declaradas

Começámos por considerar todas as regras que envolviam a declaração de variáveis em específico, daí que não tenhamos alterado a função visitDeclaration.

- **`visitTableDeclareBase()`**: gerada graças ao *subrule* #TableDeclareBase, esta regra define a variável "tableVar" (ver gramática), ou seja, é verificada antes de qualquer outra operação se a sua instância da SymTable (`sym_t`) contém uma variável com o mesmo nome e, caso afirmativo, a execução é abortada.
- **`visitTableDeclareJoining()`**: de forma análoga à regra anterior, é verificada a existência de alguma variável com o mesmo nome antes de qualquer operação.
- **`visitColumnDeclare()`**: apesar de ser separado em *subrules*, todos fazem a verificação semântica de forma semelhante: segundo a gramática, pretendemos criar uma variável com o valor de "columnVar", daí, verificamos se não existe nenhuma variável com esse valor na `sym_t`.
- **`visitColumnNameDec()`**: neste caso, queremos criar uma variável de nome igual ao valor em "itemVar" que irá guardar o nome de uma coluna. Por conseguinte, é necessária a verificação da existência da mesma como foi dito acima.
- **`visitItemDeclare()`**: novamente, mesmo sendo separado em *subrules* para operações posteriores, fundamentalmente pretende-se declarar um "itemVar" nas três regras, o que obriga a inexsitência do mesmo na `sym_t`.

Referência a variáveis não declaradas

Inferimos em que regras são chamadas variáveis cuja existência é pressuposta, maioritariamente presentes na secção de operações (delete, put, alter e export).

- **`visitTableDeclareJoining()`**: nesta regra, pretende-se juntar duas tabelas numa nova, o que obriga a definição das tabelas que se pretendem unir. Por conseguinte, verificamos se ambas existem antes de proceder à operação. Caso tal não se comprove, procedemos ao processo do erro, fornecendo mensagens claras sobre onde a operação falhou.
- **`visitColDeclareOne()/visitColDeclareTwo()`**: caso se pretenda inicializar uma coluna com base numa tabela, é verificada a existência da mesma após a imposição da regra supracitada.
- **`visitColBase()`**: precisamos de verificar a existência de todas as itemVars que poderão ser referenciadas. Se alguma não existir, o programa termina.
- **`visitColFetch()`**: quando copiamos uma coluna queremos que a coluna a copiar exista e esteja presente na SymTable.
- **`visitColumnNameDec()`**: pretendemos ir buscar o nome de uma coluna associada a uma tabela, daí que seja necessária a verificação da existência da mesma.
- **`visitItemDeclTable()`**: é de salientar o facto que acesso a itens/colunas inexistents também entra como violação desta regra. Por conseguinte, quando tentamos aceder a um indice superior ao tamanho de uma coluna ou tentamos ir buscar uma coluna não contida dentro de uma tabela é lançado um erro. Para além disso, voltamos a detetar o erro de tentar ir buscar valores a uma tabela que não existe.
- **`visitItemDeclVar()`**: neste caso, pretendemos "copiar" o valor de uma variável para uma outra. Logo, é necessário que a var a copiar tenha sido previamente declarada.
- **`visitColumnAssign()/visitColumnNameAssign()/visitItemAssign()`**: neste caso, ao tentarmos atribuir um valor a uma tabela, é pressuposta a sua declaração, daí que seja lançado um erro caso esta não tenha sido feito *a priori*.
- **`visitDeleteTable()/visitDeleteColumn()/visitDeleteItem()`**: passando agora para a secção de operações; tal como nas *callbacks* acima, para invocar métodos sobre objetos, é necessária a sua declaração. Por isso, de forma análoga, quando se tenta eliminar um elemento, é verificada a sua existência na `sym_t` (ou a existência de um sub-elemento selecionado, i. e.,

uma coluna dentro de uma tabela) e apenas se procede à eliminação *a posteriori*. O mesmo se repete para todas as operações subsequentes.

Operações sobre tipos incompatíveis

Até agora, apenas verificámos a existência (ou ausência dela) de variáveis na Tabela de Símbolos do Visitor. Contudo, há um tipo de erro que exige mais para além disso. Por exemplo, se tentarmos adicionar uma coluna a uma variável que armazena um valor do tipo `column` deveria ser lançado um erro a expressar que esta operação não está definida para um objeto deste tipo. São precisamente estas operações que serão cobertas nesta secção.

- **`visitTableDeclareJoining()`**: quando tentamos juntar duas tabelas, é certificado que as variáveis que são usadas estão associadas a um Symbol do tipo table, sendo lançado um erro caso tal não se verifique.
- **`visitColDeclareOne()/visitColDeclareTwo():`** ao usarmos uma variável como referência para inicializar uma coluna é necessário verificar que essa variável diz respeito a uma Tabela.
- **`visitColBase():`** se declararmos uma coluna usando itemVars é verificado que cada uma das variáveis referenciadas diz respeito a um Symbol do tipo item.
- **`visitColFetch():`** quando duplicamos uma coluna, isto é, criamos uma variável cujo valor é uma cópia do valor associado a uma variável já existente, verificamos se a segunda diz respeito a uma coluna.
- **`visitColumnNameDec()/:`** é certificado que "tableVar", a variável à qual iremos buscar a coluna cujo nome se pretende associar a uma variável do tipo Item, está associado a um Symbol de tipo Tabela.
- **`visitTableAssign()/visitColumnAssign()/visitColumnNameAssign()/visitItemAssign():`** atribuições de valor partem de variáveis que já foram declaradas. Por conseguinte, o seu tipo não pode ser alterado na atribuição, logo é certificado que a variável a que estamos a atribuir um valor está apta a receber este tipo de valor.
- **`visitDeleteTable()/visitDeleteColumn()/visitDeleteItem():`** como já foi mencionado acima, é necessário que cada uma destas operações seja feita sobre o tipo de dados que o nome da função sugere. Logo, quando se tenta eliminar qualquer variável é verificado o seu tipo de dados antes da ação ser concluída. O mesmo tipo de lógica é seguido ao longo das restantes operações. (put/alter)

- **`visitExportCSV():`** visto que só é possível invocar este método sobre objetos do tipo table, é verificado o tipo de "tableVar" de forma semelhante às já vistas e é abortada a execução do programa caso não seja do tipo Tabela.

Geração de Código

Feitas as análises sintática e semântica, é a função do compilador gerar código numa dada linguagem-alvo. Considerámos que o UTLCompiler deveria gerar código Java visto que é uma linguagem com a qual o grupo está bem familiarizado e que já define estruturas de dados úteis e relevantes ao tema por nós escolhido, nomeadamente HashMaps e ArrayLists.

O código é gerado através da construção de Strings que são posteriormente impressas no ficheiro generatedCode.java.

Os programas responsáveis pela geração de código são o UTLCompiler.java, que segue uma estrutura e lógica semelhante ao SemanticAnalysis, e o UTLMain.java, visto que há certas funções e sequências de caracteres que necessitam de ser gerados antes das funcionalidades do programa em si.

Utilização de UTL

Para além do *script* exemplo fornecido, seguem-se algumas instruções básicas, centrais à Unified Table Language.

Declaração Trivial de Elementos:

Itens:

```
Item i1 = "pessego"
```

Colunas:

```
Column c1 = "carros":"Volkswagen","Audi"
```

Tabelas:

```
Table t1 = ["column_one":"a","b","c";
"column_two":"alfa","beta","r"]
```

Declaração de Elementos recorrendo a variáveis:

Itens:

```
Item i2 = i1 # copia o valor de i1 para i2
Item i3 = t1.col(1).name # i3 = "column_two"
Item i4 = t1.col(1).item(0) # i4 = "alfa"
```

Colunas:

```
Column c2 = t1.col(0).item(0) # c2 =
"carros":"Volkswagen"
Column c3 = t1.col(1).item(0-2) # c3 =
"column_two":"alfa","beta","r"
Column c4 = t1.col(1) # c4 =
"column_two":"alfa","beta","r"
Column c5 = "fruta":"banana",i1
Column c6 = "fruit" -> c5 # c6 =
"fruit":"banana","pessego"
Column c7 = "empty"
```

Tabelas:

```
Table t2 = ["no_meaning":"a","b","c", "Greek":
"1","2","3"]
Table t3 = t1.join(t2) # t3 =
["column_two":"alfa","beta","r";"Greek":"1","2","3"]
```

Atribuições:

Itens:

```
i2 = "compiladores"
Item i2 = i3 # ERRO SEMÂNTICO
```

Colunas:

```
c1 = "pixar": "up", "finding nemo"
c2 = t3.col(1)
```

Tabelas:

```
t1 = ["cores":"amarelo","verde";"n":"1","2"]
```

Operações:

Eliminação de Elementos:

```
t1.col(1).item(0).del # elimina o item "1"
t1.col(1).del # elimina a segunda coluna
t1.col(0).item(0-1).del # elimina os itens "amarelo" e "verde"
t1.del # elimina a tabela t1 por completo (ver nota de script exemplo)
```

Adição de Elementos:

```
t2.put(c3,c4) # adiciona as colunas c3 e c4 à tabela t2
t2.col(0).put("d", "e") # adiciona os elementos "d" e "e" à primeira coluna da tabela t2
```

Alteração de Elementos:

```
t2.col("no_meaning").item("e").alter("f") # alteração do item "e" para "f"
```

Exportação de Tabelas para um ficheiro csv:

```
t2.csv("filename") # exporta a tabela t2 para filename.csv
```

Conclusão e Notas Finais

Concluímos este relatório e projeto com uma muito melhor noção do funcionamento de um compilador e das variadas escolhas de *design* que podem ser tomadas quando se desenvolve um. Gostaríamos também de apontar a estratégia por nós seguida ao longo de todo o projeto.

Pensámos em desenvolver um projeto simples inicialmente, mas sempre com a ideia de escalar em mente, para posteriormente conseguirmos subir o nível da UTL sem comprometer o desenvolvimento do projeto, visto que teríamos como base uma versão minimalista do mesmo. Acabámos por não adicionar tantas funcionalidades à linguagem como planeávamos, mas consideramos que conseguimos uma linguagem elegante, de sintaxe clara, minimamente robusta e com aplicações práticas, nomeadamente, a exportação de tabelas definidas dentro da linguagem para ficheiros csv.

Para além disso, apesar de não ser o objetivo do projeto, foi-nos pedido que usássemos o repositório presente no code.ua como meio de partilha de código, o que permitiu melhorar e fortalecer a nossa experiência com git e o trabalho em equipa.

Por fim, o trabalho acabou por ser feito de acordo com a seguinte percentagem:

- Camila Uachave - 10%
- João Nogueira – 23,33%
- José Frias – 23,33%
- Marta Ferreira – 10%
- Pedro Miguel – 10%
- Tiago Melo – 23,33%