Trabalhos de Laboratório de Interpretação e Compilação de Linguagens de Programação

11 de Maio de 2013

Os trabalhos práticos da disciplina de Interpretação e Compilação de Linguagens, incluindo o trabalho final, têm por objectivo a implementação de um compilador e um interpretador para uma linguagem de programação. Os trabalhos das aulas práticas são dirigidos de modo a que o trabalho final seja apenas uma extensão do trabalho realizado nas aulas práticas. As aulas práticas têm como material de apoio o conteúdo dos ficheiros disponibilizados na página da disciplina no moodle.

1 Construção de um Analisador sintáctico

Neste trabalho pretende-se construir uma calculadora simples, ou seja, um interpretador para a linguagem das expressões aritméticas. A primeira peça deste programa é um analisador sintáctico para a linguagem. Dada a sintaxe das expressões aritméticas definida por uma gramática construa o analisador sintático recorrendo a um gerador de analisadores sintáticos. Há geradores de analisadores sintáticos para várias linguagens de programação, descrevem-se de seguida as várias alternativas disponíveis.

1.1 Java

Para implementar o reconhecedor sintático em Java utiliza-se um gerador de analisadores sintácticos (LL(k)) para a linguagem Java, chamado JavaCC (Java Compiler Compiler). Pode encontrar um tutorial de JavaCC em apêndice a eeste documento. Para utilizar gramáticas LALR podem utilizar-se os geradores `cup` e `jflex`. Considere a gramática com o não terminal `E` e os terminais `"+"`, `"-"`, `"*"`, `"/"`e `num` representando literais inteiros:

```
E ::= E "+" E | E "-" E | E "*" E | E "/" E | "("E")" | num
```

Note que esta gramática é ambígua, ou seja, para uma mesma expressão é possível desenhar mais do que uma árvore de derivação sintáctica. É portanto necessário desambiguar a gramática tornando explícitas as prioridades dos operadores. Pode-se proceder a modificações na gramática para conseguir este fim. Considere a gramática equivalente não ambígua com os não terminais `E`, `F`, e `T`:

```
E ::= T ( "+" | "-" ) E | T
T ::= F ( "*" | "/" ) T | F
F ::= num | "(" E ")"
```

Esta gramática aceita a mesma linguagem que a anterior mas tem como resultado possível apenas uma derivação sintática por cada frase. Utilize como base a gramática JavaCC definida no ficheiro `Calc.jj` fornecido e construa um analisador sintático que aceite a linguagem CALC definida pela gramática acima. Note que no ficheiro da especificação da gramática o não terminal `E` é representado pela produção `exp()`, o não terminal `T` é representado pela produção `term()` e o não terminal `F` é representado pela produção `factor()`.

Utilize a classe Calc, definida no ficheiro Calc.java que lê uma expressão do Objecto System.in (a consola) e faz uso da classe Parser produzida mecanicamente pelo JavaCC para validar as expressões.

Teste o seu analisador sintático com diferentes expressões aritméticas (válidas e não válidas).

1.2 OCaml

Em OCaml deve-se usar os utilitários `ocamlyacc` e `ocamllex` de acordo com o código exemplo distribuído.

2 Árvore de Sintaxe Abstracta e Interpretador

Complete a gramática JavaCC definida no ficheiro G2.jj disponibilizado de modo a que o analisador sintático construa a árvore de sintaxe abstracta (AST) da linguagem CALC que se define indutivamente da seguinte forma:

$$\begin{array}{ll} \textbf{num}: & \mathit{Integer} \rightarrow \mathit{CALC} \\ \textbf{add}: & \mathit{CALC} \times \mathit{CALC} \rightarrow \mathit{CALC} \\ \textbf{sub}: & \mathit{CALC} \times \mathit{CALC} \rightarrow \mathit{CALC} \\ \textbf{mul}: & \mathit{CALC} \times \mathit{CALC} \rightarrow \mathit{CALC} \\ \textbf{div}: & \mathit{CALC} \times \mathit{CALC} \rightarrow \mathit{CALC} \end{array}$$

Utilize para o efeito as seguintes classes e interface de exemplo:

```
IASTNode.java
ASTNum.java
ASTMul.java
Calc.java
```

Implemente as restantes classes necessárias para completar o exercício de construir a AST. Note que todas estas classes devem implementar o interface IASTNode.

Para finalizar, implemente a semântica operacional da linguagem no método evaluate das classes da AST dadas e crie as classes adicionais necessárias para avaliar qualquer expressão da linguagem CALC. Modifique o método main da

classe Calc para que este receba a AST construída, avalie a expressão e apresente o resultado.

2.1 OCaml

A alternativa é definir a AST como um tipo de dados indutivo `ast` e uma função `eval` definida indutivamente na estrutura das expressões.

3 Linguagem com identificadores (CALCI)

O objectivo é desenvolver o interpretador para a linguagem CALCI, uma extensão da linguagem CALC com identificadores e com a expressão de declaração de identificadores. A definição de referência pode ser encontrada no ficheiro `CALCI.hs` na página da cadeira.

1. Implemente declarações de um único identificador de cada vez. A sintaxe concreta da construção de declaração é `decl x = 1 in E end`. A sintaxe abstracta da linguagem é a seguinte:

num :	$Integer \rightarrow CALCI$
id :	$String \rightarrow CALCI$
add :	$CALCI \times CALCI \rightarrow CALCI$
sub :	$CALCI \times CALCI \rightarrow CALCI$
mul :	$CALCI \times CALCI \rightarrow CALCI$
div :	$CALCI \times CALCI \rightarrow CALCI$
decl :	$String \times CALCI \times CALCI \rightarrow CALCI$

   Implemente um interpretador com ambientes referindo-se à técnica explicada nas aulas teóricas da disciplina. Deve modificar o interface `IASTNode` para que a operação `evaluate` aceite um ambiente como parâmetro. Defina a classe `Env` implementando o interface genérico

   ```
   interface IEnv {
     IEnv beginScope();
     IEnv endScope();
     void assoc(String id, int v)
     int find(String id);
   }
   ```

   Note que um objecto ambiente é gerado por um constructor vazio (o primeiro ou mais exterior) ou por uma chamada ao método `beginScope` do ambiente imediatamente exterior. O método `find` de um ambiente procura um identificador nas suas declarações locais e depois nos ambientes mais exteriores. O método `find` pode lançar uma excepção `IdentifierNotFound` no caso de não ter ambiente exterior e não encontrar o identificador pedido.

2. Implemente declarações múltiplas (`decl x = 1 y=2 in E end`) com a seguinte sintaxe abstracta:

$$
\begin{array}{ll}
\textbf{num}: & Integer \rightarrow CALCI \\
\textbf{id}: & String \rightarrow CALCI \\
\textbf{add}: & CALCI \times CALCI \rightarrow CALCI \\
\textbf{sub}: & CALCI \times CALCI \rightarrow CALCI \\
\textbf{mul}: & CALCI \times CALCI \rightarrow CALCI \\
\textbf{div}: & CALCI \times CALCI \rightarrow CALCI \\
\textbf{decl}: & List(String \times CALCI) \times CALCI \rightarrow CALCI
\end{array}
$$

3.1 OCaml

A interpretação em OCaml é baseada num ambiente implementado com uma lista de associações identificador, valor. No caso de declarações múltiplas, utilize listas de listas de associações.

3.2 Haskell

Como já foi observado no início a implementação Haskell (`CalcI.hs`) serve de referência para este trabalho. Baseia-se num Monad `Maybe Int` para capturar a possibilidade de erro de execução.

4 Linguagem Imperativa (CALCState)

Nota: No ano de 2012/2013 foi implementada a linguagem CALCStatedas aulas teóricas.

O objectivo deste trabalho é estender a linguagem CALCI com operacões imperativas e definir a linguagem CALCSTATE. A sintaxe da linguagem é a seguinte:

```
E ::= E + E
  | id
  | num
  | true
  | false

  | E - E
  | E * E
  | E / E
  | ( E )
  | E = E
  | E > E
  | E & E
  | not E
```

```
| var(E)
| !E
| E := E

| while E do E end
| if E then E else E end
| E ; E
| decl id = E in E end
| print(E)
| println()
```

Estenda o *parser* desenvolvido no trabalho prático nº4 com estes novos operadores.

Implemente de seguida um interpretador para a linguagem CALCSTATE, adaptando o modelo de ambiente e memória apresentado nas aulas teóricas e a função de interpretação semântica respectiva da forma que achar mais eficiente.

> **Nota:** Implemente um interface para representar o tipo dos valores (IValue) e classes Java implementando esse interface (subtipos) para representar os vários tipos de valores manipulados pelo interpretador, da forma requerida pela semântica da linguagem: Inteiros, Booleanos e Referências (IntValue, BoolValue e RefValue).
>
> **Nota:** Sugere-se a implementação das células de memória através de objectos com métodos set e get. Utilize os valores do tipo referência como células de memória.
>
> **Atenção:** Não implemente para já nenhum mecanismo de detecção de erros de execução, deixe que o programa interpretador pare a execução de um programa com uma excepção não tratada!
>
> Por exemplo, o programa
>
> ```
> decl x = var(2) in
> decl y = 3 in
>   print (x+y); println()
> end
> end
> ```
>
> "estoira"ao executar a operação de adição, pois o identificador **x** não denota um inteiro e o interpretador não suporta operações entre inteiros e referências. Por outro lado, o programa
>
> ```
> decl
>   x = var(0)
> in
>   while (100 > !x) do
>     x := !x + 1;
> ```

```
      print ( !x ); println()
    end
  end
```

imprime todos os números de 1 a 100.

5 Abstração Funcional (Core)

Considere a linguagem imperativa Core, extendida com as construções que suportam abstração funcional: a definição de uma função e a chamada de uma função.

```
E ::= E + E
  | id
  | num
  | true
  | false

  | E - E
  | E * E
  | E / E
  | ( E )
  | E = E
  | E > E
  | E & E
  | not E

  | var(E)
  | !E
  | E := E

  | while E do E end
  | if E then E else E end
  | E ; E
  | decl id = E in E end
  | print(E)
  | println()

  | fun x -> E end
  | E (E)
```

Neste trabalho implemente o interpretador da linguagem Core. A sintaxe e semântica de referência em relação à abstração funcional encontra-se na página da cadeira no ficheiro `CalcF.hs`

6 Verificação de tipos (Core)

Considere a linguage Core com a seguinte extensão:

```
E ::= ...

  | fun x : T -> E end
  | E (E)

T ::= int | bool | ref(T) | funT(T,T)
```

Implemente um sistema de tipos para a linguagem Core tendo como referência a especificação em `CalcFT.hs` e `CalcStateT.hs`.

O objectivo deste trabalho é desenvolver o sistema de tipos para as linguagens CALCF e CALCSTATE. O sistema de tipos deve ser integrado quer no interpretador quer futuramente no compilador. A sintaxe da linguagem mantém-se inalterada em relação ao problema anterior excepto na definição de funções.

O sistema de tipos é implementado através de mais uma função de interpretação semântica dos programas. Note que os tipos possíveis na linguagem `CALCSTATE` são os inteiros, os booleanos, as referências e as funções. Os tipos de um programa CALCSTATE são valores do tipo $TYPE$ definidos indutivamente com os constructores:

$$\begin{array}{ll} \textbf{int}: & \textbf{void} \rightarrow TYPE \\ \textbf{bool}: & \textbf{void} \rightarrow TYPE \\ \textbf{ref}: & TYPE \rightarrow TYPE \\ \textbf{fun}: & TYPE \times TYPE \rightarrow TYPE \end{array}$$

> **Nota:** Essa função deve ser implementada como um método das classes que compõem a AST chamado `typchk` tendo como parâmetros um ambiente (contendo uma relação entre identificadores e tipos) e tendo o tipo resultado um valor do domínio dos tipos.
>
> **Nota:** Para definir o domínio dos tipos, defina um interface `IType` e implemente cada um dos constructores dos seus valores com classes derivadas.
>
> **Nota:** Implemente a função `equals` inductivamente na estrutura dos tipos.
>
> **Nota:** Em caso de erro, a função deve terminar com uma excepção própria.

7 Compilação de linguagens de expressões (CALC)

(Enunciado resumido: estes enunciados serão completados progressivamente e a informação relevante será também passada em aula).

Implemente um compilador da linguagem CALCpara LLVM. Referência em `CalcComp.hs`.

8 Compilação da linguagem CALCI

Implemente um compilador da linguagem CALCIpara LLVM. Referência em `CalcComp.hs`.

9 Compilação da linguagem CALCState

Implemente um compilador da linguagem CALCStatepara LLVM. Referência em `CalcStateComp.hs`.

10 Compilação da linguagem Core

Implemente um compilador da linguagem CALCStatepara LLVM.

11 Projeto Final

Implemente um interpretador e um compilador para a linguagem CALCState, com abstração funcional, e construções imperativas

11.1 Pontos de Bónus

Declarações recursivas e gestão de memória automática.

A Tutorial de JavaCC

JavaCC = Java Compiler Compiler

Este apêndice pretende ilustrar a utilização da ferramenta JavaCC para construir analisadores sintácticos que servem de suporte aos processos de compilação / execução das linguagens de programação. A ferramenta escolhida para a geração automática dos analisadores sintácticos foi o JavaCC (Java Compiler Compiler). O JavaCC é um gerador de analisadores sintácticos para a linguagem Java. O JavaCC é semelhante ao Yacc pois gera um analisador sintáctico em Java a partir de uma gramática em notação EBNF. O JavaCC gera um analisador recursivo descendente limitando o seu uso a gramáticas LL(k). Um parente mais próximo dos clássicos Lex e Yacc para a linguagem Java (que geram parsers LALR) são o JFlex e o Cup.

A.1 Primeiro Programa

Neste primeiro exemplo encontramos uma gramática que reconhece expressões começadas (e terminadas) pela letra "a"e que entre estas duas letras tem um número não nulo de letras b. Considere-se o seguinte ficheiro de input para o JavaCC:

```
PARSER_BEGIN(Parser)
  public class Parser{
    public static void main(String args[])  {
      Parser parser = new Parser(System.in);
      for (;;)    {
        System.out.print("> ");
        try      {
          parser.ABA();
          System.out.println("Ok!");
        }  catch (ParseException x) {
          System.err.println("Syntax Error.");
          parser.ReInit(System.in);
        }
      }
    }
  }
PARSER_END(Parser)

SKIP : {  " " |
              "t"
}

TOKEN :{
    < A:   "a" >
  | < B:   "b" >
```

```
  | < EOL: "n" >
}

void ABA() :{}{  <A> ( <B> )+ <A>}
```

Neste ficheiro é possível observar a declaração de um conjunto de símbolos terminais (tokens), símbolos não-terminais (neste caso apenas `ABA`) e de regras. Note que para definir completamente uma gramática apenas nos falta indicar qual o símbolo inicial da gramática e o conjunto de símbolos terminais (ou, neste caso, condições de terminação). O JavaCC gera um conjunto de classes, enriquecendo uma delas (a indicada nas expressões `PARSER_BEGIN` e `PARSER_END`) com métodos correspondentes a cada uma das regras da gramática (note que o cabeçalho das regras segue a sintaxe do Java, seguido de ":" . O conjunto de símbolos não-terminais corresponde ao conjunto dos nomes destas regras (métodos). Podemos escolher o símbolo inícial que desejarmos, bastando para isso chamar o método com o seu nome. Os símbolos terminais são definidos na secção TOKEN. É ainda possível definir um conjunto de elementos a ignorar durante a análise (secção SKIP). Note-se, finalmente, que a regra faz uso dos símbolos terminais e do modificador + (com a semântica habitual de repetição, um ou mais) para a sua definição.Ao compilarmos com o JavaCC esta gramática (convenientemente colocada num ficheiro chamado ABA.jj):

```
$ javacc ABA.jj
```

obtemos uma série de classes Java que depois podemos compilar e correr

```
$ javac Parser.java
$ java Parser
```

Este programa aceita agora palavras válidas na linguagem especificada. O output é "`Ok`"! se a palavra pretence à linguagem e um erro se tal não acontecer.

subsectionSegundo Programa

Neste segundo exemplo acrescentámos a hipótese de as expressões começarem (e terminarem) com a letra `b` e de entre estas aparecer um número arbitrário (possivelmente nulo) de letras `a`. Pretendemos ainda contar o número de letras `b` na expressão:

```
PARSER_BEGIN(Parser)
public class Parser{
  public static void main(String args[])  {
    Parser parser = new Parser(System.in);
    for (;;)    {
      System.out.print("> ");
      try {
        int n = parser.Begin();
        System.out.println("Ok [B = "+n+"]!");
      } catch (ParseException x) {
```

```
        System.err.println("Syntax Error.");
        parser.ReInit(System.in);
      }
    }
  }
}
PARSER_END(Parser)

SKIP :{ " "
          | "t"}

TOKEN :{
     < A:   "a" >
   | < B:   "b" >
   | < EOL: "n">}

int Begin() :{ int count; }{
  count = ABA() <EOL> { return count; }
  |
  count = BAB() <EOL> { return count; }
}

int ABA() : { int coun

t = 0; }{
  <A> ( <B> { count++; } )+ <A> { return count; }
}

int BAB() :{}{
  <B> ( <A> )* <B> { return 2; }
}
```

Note que neste programa associámos acções às regras. Basta para isso colocar código Java entre chavetas em qualquer parte da regra. Este código é executado sempre que o analisador passar por essa parte da regra. Note ainda que desta feita as regras devolvem valores, e que é possível guardar esses valores em variáveis. Existe uma secção indicada para a declaração dessas variáveis, essas declarações deverão ser efectuadas dentro do primeiro par de chavetas das regras.