INSTITUTO MAUÁ DE TECNOLOGIA

MAUÁ

ECM253 – Linguagens Formais, Autômatos e Compiladores

Projeto

Analisador CUP com ações semânticas

Marco Furlan

Outubro de 2024

1 Modificações na sintaxe do analisador de expressões simples

Alterar o projeto do **interpretador** de expressões simples com **CUP**, apresentado na **aula passada**, para que um **programa** seja uma **sequência** de **comandos** (não produzem valor) e que manipulem **expressões** (estas produzem valor). **Adicionar**/**alterar** os seguintes **elementos**:

(a) **Utilizar apenas números reais**. Será necessário alterar no analisador léxico do JFlex a expressão regular que reconhece o número e depois no CUP.

(b) **Adicionar novos comandos**. Os **comandos** serão:

(i) **Comando de atribuição**: o comando de atribuição deverá possibilitar o uso de **variáveis** no programa. **Exemplo de uso**:

```
x = 10.5;
y = 5.0;
z = x + y;
```

Para que se possa obter o valor de uma variável, uma variável sozinha deverá ser considerada uma **expressão** (veja na gramática).

(ii) **Comando de saída na tela**: criar o **comando** `print(e)` para exibir **valores de um expressão** `e`. **Exemplo de uso**:

```
print(z);
print(7/3);
```

(c) **Adicionar novos operadores**. Os **novos operadores** serão:

(i) **Operador de exponenciação** (`**`): trata-se de um **operador** cuja **precedência** é **maior** do que a **precedência** de **multiplicação ou divisão**, mas **menor que inversão de sinal**. Sua **associatividade** deve ser da **direita para a esquerda**. **Exemplo**:

```
y = x ** 3;
```

Será necessário alterar tanto o analisador léxico e sintático (pense!).

(ii) **Símbolo `PI`** (π): quando usado, retornará o valor de π. Será necessário alterar tanto o analisador léxico quanto o sintático (pense!). **Exemplo**:

```
y = sin(PI/3);
```

(iii) **Adicionar** as **funções matemáticas** `sin(x)` e `cos(x)`, onde `x` é **qualquer expressão** da gramática. Utilizar valores em **radianos**. **Exemplo**:

```
y = cos(PI/3);
```

(d) **Definir uma variável como expressão**. É importante tratar uma **variável** como uma **expressão**, para **recuperar** o seu **valor** quando **necessário**. **Exemplo**:

```
y = x ** 3;
```

Neste caso, a **variável** `x`, ao ser **referenciada**, retornará seu valor (da tabela de variáveis do ambiente – veja na seção 2.1) para ser depois elevado ao valor três.

A **gramática** que deverá ser **implementada** no **CUP** está apresentada a **seguir**:

```
program ::= command_list

command_list ::= command_list command_part
    | command_part

command_part ::= command ';'

command ::= assignment_command
    | print_command

assignment_command ::= id '=' expr

print_command ::= 'print' '(' expr ')'

expr ::= expr '+' expr

expr ::= expr '-' expr

expr ::= expr '*' expr

expr ::= expr '/' expr

expr ::= expr '%' expr

expr ::= expr '**' expr

expr ::= '-' expr

expr ::= '(' expr ')'
```

```
expr ::= number

expr ::= id

expr ::= 'sin' '(' expr ')'

expr ::= 'cos' '(' expr ')'

expr ::= 'PI'
```

Nesta gramática, `id` deverá ser uma **expressão regular** definida assim:

```
id = [A-Za-z_][A-Za-z_0-9]*
```

E `number` é uma **expressão regular** que casa com um número real, definida assim:

```
\d+(\.\d+)?(["E""e"]["+""-"]?\d+)?
```

Onde `\d` é uma expressão regular do JFlex que casa com um dígito. Analisar nos códigos quais outros locais que será necessário modificar.

2 Semântica dos elementos do programa

2.1 Ambiente de variáveis

Para utilizar **variáveis**, **implementar** um **sistema** de **ambiente** para o **interpretador**. Ambiente é o **nome** dado à **área de memória** do **programa** onde serão **armazenadas** as **variáveis durante a execução**.

Uma **forma eficiente** para **armazenar variáveis e seus valores** correspondentes é utilizar uma **tabela de hash**. Em Java, a classe `HashMap` permite criar uma **tabela de variáveis** de modo a armazenar as variáveis durante a execução do programa.

Para fazer isso no **CUP**, modificar `Parser.cup`, adicionando a linha a seguir na parte de importações:

```
import java.util.HashMap;
```

Adicionar uma **seção de código** para declarar um objeto do tipo `HashMap` na classe `Parser`, que será utilizado como **tabela de símbolos**. Ele deve **mapear** nomes de variáveis à números. Adicionar esta seção logo após as importações de pacotes:

```
parser code {:
    // symbolTable é a tabela de símbolos
    private HashMap<String, Double> symbolTable = new HashMap<>();
:}
```

2.2 Atribuição e recuperação de valores de variáveis

Para que a **atribuição** de um **valor** à **variável** tenha **efeito**, **alterar o comando de atribuição** no **CUP** para que, quando houver uma atribuição, seja **adicionado à tabela de símbolos** o **texto com o nome do identificador** e o **valor** sendo atribuído. **Não será necessário retornar nada no comando de atribuição**.

Será necessário **adicionar** uma **regra** à **gramática** para **recuperar valores** de variáveis **e permitir** que elas **sejam utilizadas em expressões**. Basta **consultar o valor** de uma **variável** na **regra** que define que uma **expressão** é uma **variável** e então obter seu valor da tabela.

2.3 Funções matemáticas e `PI`

Basta **chamar** as **funções Java** da classe `Math` como **ação semântica nas regras** adequadas (`sin()`, `cos()`, `pow()` e `PI`).

2.4 Comando de impressão na tela

Basta apenas **executar** um **comando Java** para **exibir valor** na **tela** como **ação semântica** na **regra** que analisa o **comando** `print()`, exibindo o valor da expressão.