INSTITUTO MAUÁ DE TECNOLOGIA

MAUÁ

ECM253 – Linguagens Formais, Autômatos e Compiladores

Projeto

Analisador CUP com comandos de decisão, escopo e tipos

Marco Furlan

Novembro de 2024

1 Introdução

O objetivo deste projeto é modificar o projeto com comandos de decisão e repetição presentes no arquivo `Atividade - comandos de decisão, tipos e escopos.zip`, apresentado em aula.

A gramática deste projeto está apresentada a seguir (ver sua implementação no arquivo do projeto):

```
program ::= command_list

command_list ::= command_list command_part
    | command_part

command_part ::= command

command ::= assignment_command ';'
    | print_command ';'
    | if_command
    | while_command

while_command ::= 'while' '(' bool_expr ')' command

assignment_command ::= id '=' expr

print_command ::= 'print' '(' bool_expr ')' command

if_command ::= 'if' LPAREN '(' bool_expr ')' command

expr ::= expr '+' expr
    | expr '-' expr
```

```
    | expr '*' expr
    | expr '/' expr
    | expr '%' expr
    | expr '**' expr
    | '-' expr
    | '(' expr ')'
    | number
    | id
    | 'sin' '(' expr ')
    | 'cos' '(' expr ')
    | pi

bool_expr ::= expr '>' expr
```

Lembrar que nesta gramática:

- `number` é uma expressão regular que casa com um número real.
- `id` é uma expressão regular que casa com um identificador.
- `pi` é uma expressão regular que casa com o valor de π.

A seguir tem-se as modificações que é para fazer nesta tarefa.

2 Completar o comando condicional

Adicionar na linguagem a **cláusula** `else`, para completar o comando `if`. Neste caso, é necessário **alterar** tanto o **analisador léxico** quanto o **analisador sintático**:

- No **JFlex** adicionar a palavra `else` e retornar um símbolo para ela.
- No **CUP** adicionar o símbolo que será retornado pelo **JFlex**.

Depois, para **evitar** a **ambiguidade** do **comando** `if`, declarar tanto `if` quanto `else` como **operadores** (de mais baixa precedência possível) onde `if` é **tem maior precedência** do que `else`:

```
// Precedência e associatividade dos operadores
precedence nonassoc ELSE;
precedence nonassoc IF;
...
```

Entãp, **alterar** a **gramática** e as **árvores AST** para incluir o `else`. A **gramática original em BNF** deverá ficar assim:

```
    ...
    if_command ::= 'if' LPAREN '(' bool_expr ')' command
        | 'if' LPAREN '(' bool_expr ')' 'else' command
    ...
```

Notar que, no **CUP**, é **necessário** adicionar **suporte** ao **processamento semântico** de "else". Atualmente está assim:

```
...
if_command ::= IF LPAREN bool_expr:e RPAREN command:c
        {:
            RESULT = new IfCommand(e, c);
        :}
    ;
...
```

Para **adicionar o processamento** do "else" basta:

(i) Adicionar a **opção** da **regra** `if_command` para tratar o comando `completo` "if-then-else" (veja no BNF anterior);

(ii) Adicionar um **novo campo** e um **novo construtor** na **classe** `IfCommand` de modo a **armazenar** uma **referência** para um **comando** a ser executado na parte "else" do comando, se ela existir, e deixar este campo com **valor nulo** se o **construtor original** de "if" for utilizado;

(iii) **Alterar** a **operação** `visit()` que visita o comando "if" para também visitar o comando "else".

Depois de fazer essas alterações testar o programa:

```
x = 1;
if (x>4)
    print(0);
else
    print(1);
```

Deverá ser apresentado o valor `1` na tela.

3 Adição de bloco de comandos

Para se poder **executar mais** de **um comando** dentro de um **comando** "if-then-else" ou "while", **adicionar** o **conceito** de **bloco**. A **gramática original** deverá ser **alterada** para:

```
...
command ::= ...
    ...
    | command_block

command_block ::= '{' command_list '}'
...
```

Para isso, alterar o arquivo **JFlex** e **CUP** haver o reconhecimento dos símbolos "{" e "}". Depois alterar o projeto assim:

(i) **Criar** no **pacote** `command` a **classe** `CommandList`, implementando a classe `Command` e que armazenará referências a comandos, formando uma lista de comandos:

```
public class CommandList implements Command {
    public CommandList commandList;
    public Command command;

    public CommandList(Command command, CommandList commandList) {
        this.commandList = commandList;
        this.command = command;
    }

    public CommandList(Command command) {
        this(command, null);
    }

    @Override
    public void accept(CodeVisitor v) {
        v.visit(this);
    }

}
```

(ii) **Não esquecer** de **alterar** convenientemente a **interface** `CodeVisitor` e, na **classe** `Interpreter`, **escrever** o **código** que **interpreta** a **lista de comandos**:

```
    // ...
    @Override
    public void visit(CommandList commandList) {
        CommandList cl = commandList;
        do {
            cl.command.accept(this);
            cl = cl.commandList;
        } while (cl != null);
    }
    //...
```

(iii) No **CUP**, **associar** ao **não terminal** `command_list` o tipo `CommandList`:

```
...
non terminal CommandList command_list;
...
```

(iv) No **CUP associar** ao **não terminal** `command_part` a classe `Command`:

```
...
non terminal Command command_part;
...
```

(v) No **CUP**, **alterar** a **semântica** do **não terminal** `command` para que agora ele apenas retorne um comando simples que foi reconhecido, sem interpretá-lo:

```
...
command_part ::= command:c
        {:
            RESULT = c;
        :}
    ;
...
```

(vi) No **CUP**, **alterar** a **semântica** do **não terminal** `command_list` para que agora ele apenas retorne ou uma parte de comando ou uma parte de comando seguido de uma lista de comandos (foi invertida a ordem da gramática original para facilitar o processamento em ordem dos comandos):

```
...
command_list ::= command_part:p command_list:c
        {:
            RESULT = new CommandList(p, c);
        :}
    | command_part:p
        {:
            RESULT = new CommandList(p);
        :}
    ;
...
```

(vii) No **CUP**, **alterar** a **semântica** do **não terminal** `program` para que agora ele interprete a lista de comandos que o constitui:

```
...
program ::= command_list:cl
        {:
            cl.accept(new Interpreter());
        :}
    | error
;
...
```

(viii) No **CUP**, acrescentar o **não terminal** `command_block`, que deverá ter o tipo `CommandList`, criado anteriormente:

```
...
non terminal CommandList command_block;
...
```

(ix) No **CUP**, acrescentar a **regra** que implementa o **não terminal** `command_block`:

```
command_block ::= LBRACE command_list:cl RBRACE
    {:
        RESULT = cl;
    :}
;
```

Aqui `LBRACE` e `RBRACE` são os nomes de "{" e "}", respectivamente.

(x) Por fim, no **CUP acrescentar** a **regra** que **especifica** que um **bloco de comando** é um **comando**:

```
command ::= ...
    ...
    | command_block:cb
        {:
            RESULT = cb;
        :}
    ;
```

Testar: o programa apresentado a seguir deverá escrever os valores `3` e `4` na tela:

```
x = 1;

if (x>4) {
    print(1);
    print(2);
}
else {
    print(3);
    print(4);
}
```

4 O que é para fazer e pontuação

- **Se** entregou **apenas** o **projeto desenvolvido até** este **ponto** (seguindo o **tutorial**): **6,0 pontos**.
- **Se implementar** os **operadores relacionais** "==" (igualdade), "<" (menor que), ">=" (maior ou igual), "<=" (menor ou igual) e "!=" (diferente de): **até 1,5 ponto**.
- **Se implementar escopos** com **pilha** de **tabelas** (como já visto): **até 1,0 ponto**.
- **Se implementar tipos** (apenas números reais e cadeias de caracteres), **como** já **visto em aula**: **até 1,5 ponto**.