INSTITUTO MAUÁ DE TECNOLOGIA

MAUÁ

ECM253 – Linguagens Formais, Autômatos e Compiladores

Projeto

Analisador CUP com Tipos e Escopos

Marco Furlan

Outubro de 2024

1 Objetivo

O objetivo deste projeto é implementar um sistema de tipos e escopos em um projeto já desenvolvido anteriormente. Para simplificar, não se utilizará aqui o conceito de árvore AST, sendo que as ações emânticas serão realizadas diretamente no próprio arquivo CUP.

2 Gramática original

Será utilizada como **base** a **gramática** apresentada a **seguir**, que está **implementada** no **projeto presente** no **arquivo** `SimpleExpr-VSCode.zip` anexo a esta atividade:

```
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'
```

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;

3 Gramática modificada

Pode-se **alterar** a **gramática** definida anteriormente para **trabalhar** com **dois tipos** de **dados**: **reais** (como apresentado anteriormente) e **cadeia de caracteres**. Além disso, pode-se **adicionar o conceito de escopo**, por meio da adição de **bloco de comandos**. A **versão modificada** da linguagem está **apresentada a seguir** (em **negrito** estão as **modificações** feitas na gramática original):

```
program ::= command_list
command_list ::= command_list command_part
    | command_part
command_part ::= command ';'
    | command_block
command_block ::= '{' command_list '}'
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 ::= string
expr ::= id
expr ::= 'sin' '(' expr ')'
expr ::= 'cos' '(' expr ')'
expr ::= 'PI'
```

Onde:

- `string`: é uma expressão regular que casa com uma cadeia de caracteres. É qualquer texto contendo um ou mais caracteres quaisquer, delimitado pelos símbolos " e ". **Exemplo**: `"O rato roeu a roupa do rei de Roma."`.

4 O que é para fazer?

Seguindo a **gramática** da **seção** 3, **implementar** os **requisitos** apresentados a **seguir**:

(R1) **Adicionar** novas **regras** que **possibilitem** reconhecer tanto **cadeias de caracteres** quanto **blocos de comandos** (inclusive com aninhamentos). Já está descrito na gramática apresentada na seção 3.

(R2) **Implementar** um **suporte** para **inicialização** de **cadeias** de **caracteres**, isto é, além de permitir a inicialização de variáveis por números reais, o interpretador deverá permitir a inicialização de cadeias. **Exemplo**:

```
x = 6.02E23;
s = "Bom dia!";
```

(R3) **Adicionar a semântica de tipos às expressões**. Toda **expressão** deverá ter uma **informação** que identifica seu **tipo**. O **tipo** deverá ser **obtido** a partir das **expressões constantes**. Veja na gramática:

```
...
expr ::= number
expr ::= string
...
```

Quando essas **regras** forem **reconhecidas**, é um bom **momento** para se **salvar** alguma **informação** de **tipo** (constantes comuns ou enumeradas) em **algum campo** das **expressões**.

(R4) **Utilizar o conceito de tipo para evitar confusões na mistura de tipos nas expressões**. A semântica das expressões deverá ser a seguinte:

(i) **Qualquer operação aritmética deverá ser realizada apenas entre números reais** – a violação desta regra deverá resultar em uma exceção em tempo de execução (veja uma exceção a seguir).

(ii) **No caso apenas da expressão de soma, ela admitirá a soma entre duas cadeias de caracteres**, cuja **semântica** deverá ser de **concatenar as cadeias envolvidas**, resultando em uma nova **cadeia**.

(iii) No caso **apenas da expressão de soma**, ela também **admitirá a soma** entre uma **cadeia e um número real** (e vice versa), cuja **semântica** será de **converter o número real em cadeia** de caracteres e **depois concatenar** com a outra cadeia.

(iv) **O comando `print()` deverá apresentar na tela tanto expressões numéricas quanto expressões com cadeias de caracteres**. Por **exemplo**, ele deverá **apresentar mensagem**: `"O número de Avogadro é 6.02E23"` se for assim **executado**: `print("O número de Avogadro é: "+ 6.02E23)` () (notar que a soma de número com cadeia resulta em cadeia, que depois é exibida na tela).

(v) **Não há restrição sobre o comando de atribuição**: se uma **cadeia** de **caracteres** for atribuída à uma **variável numérica**, esta **passa** a ter o **tipo** de **cadeia** de **caracteres** e **armazenar** o **valor** da **cadeia**.

(R5) Implementar o conceito de escopo:

(i) As **chaves** ({ e }) **criam** um novo **escopo** no **programa**. Quando se **utiliza** o **escopo global** (isto é, fora de qualquer par de chaves), as **variáveis** devem **durar toda** a **execução** do **programa**;

(ii) Quando se **entra** em um **novo escopo**, uma **nova tabela** de **variáveis** deve ser **criada** e **empilhada** sobre a **tabela atual**, e ficará no **topo** da **pilha** de modo que as **variáveis** no **escopo mais interno estejam** sempre na **tabela do topo**, enquanto que as **variáveis** do **escopos** mais **externos** ficam **abaixo** do **topo**;

(iii) **Se** em um **escopo mais interno** um **símbolo** for **referenciado** e **não está presente** em sua **tabela do escopo atual**, deve-se **procurar** tal **símbolo nas tabelas abaixo do topo**. **Se não for encontrado**, deve-se **produzir um erro**. **Se for encontrado**, basta **utilizar seu valor**;

(iv) A **dinâmica** para isso pode ser assim **resumida**: uma **pilha** de **tabelas** é **criada** quando o **programa** for **executado** e as variáveis do **escopo global** são **armazenadas nela** (tipo, nome e valor). **Enquanto não** se entrar em **nenhum escopo**, **utiliza-se esta tabela** para as **variáveis** do **escopo atual**. Quando o **interpretador encontrar** "`{`", ele prontamente **cria** uma **nova tabela** e **adiciona** ao **topo** da **pilha** (operação `push()`);

(v) Qualquer **referência** a uma **variável** sempre **leva** à **consulta** da **tabela** que está no **topo** da **pilha**. Se **não** for **localizada** nesta **tabela**, **verifica-se** se existe **alguma tabela abaixo** (sem desempilhar!!!) do **topo** e, assim por diante. **Se passar** por **todas** as **tabelas**, do **escopo atual até o mais externo (base da pilha)**, e **não** for **localizado** o **elemento**, produzir uma **mensagem** de **erro** e **abortar** a **operação**. **Caso contrário**, **obter** o **valor** da **variável** e usá-lo;

(vi) Para **acessar** a **tabela** do **topo** da pilha **sem removê-la**, utilizar a **operação** `peek()`.

(vii) Quando o **interpretador encontrar** o **símbolo** "`}`" ele **remove** a **tabela** do **topo** (**operação** `pop()`) e **segue** em **frente** com as **computações**.

(viii) **Sugestões**: para **implementar** a **pilha** pode-se utilizar a **classe** `Stack` e para **criar** uma **tabela** de **variáveis**, utilizar a **classe** `HashMap`. Para **percorrer** as **tabelas** da **pilha**, utilizar um `Iterator`.

(ix) Por fim, **toda vez que se alterar** uma **variável** em um certo **escopo** é **preciso atualizá-la na tabela**.

5 Exemplo de um programa

```
msg = "Olá Mundo!";
print(msg);
x=10.5;
print("O valor de x é: " + x)
{
    print("Entrando em um escopo...");
    print(x);
    x = 7.4;
    print(x);
    print("Saindo de um escopo...");
}
```