Universidade Federal do Rio Grande do Norte
Departamento de Informática e Matemática Aplicada

Disciplina: DIM0611 — Compiladores
Docente: Martin Alejandro Musicante
Discente: Felipe Cortez de Sá

Relatório — Analisador léxico

1 A linguagem

1.1 Tipos

1.1.1 `integer`

Um número positivo, negativo ou zero, ou seja, inteiro. Decidiu-se tratar o operador unário `-` como token separado do número. Assim, a expressão regular para um inteiro é simplesmente `[0-9]+`.

1.1.2 `number`

Um número em ponto flutuante. Não há suporte a notação científica. Dessa forma, a expressão regular associada ao token é `[0-9]+"."[0-9]*`

1.1.3 `boolean`

Pode assumir apenas os valores `true` ou `false`. Logo, para o token booleano, temos a expressão regular `"true"|"false"`

1.1.4 `text`

Uma string. Como é não-trivial tratar as quebras de linha e o que deve vir dentro das aspas, foi retirada deste site uma expressão regular para tokens de strings. A expressão regular extraída e adaptada é `\"(\\.|[^\\"])*\"`

Para a *BNF*, temos então os tipos `type ::= "integer" | "number" | "boolean" | "text"`

1.1.5 Conversão entre tipos

A conversão entre tipos se dá através da função `cast(tipo, identificador)`.

1.1.6 Identificadores

Devem começar com uma letra seguida por zero ou mais combinações entre números, letras ou underscores. Temos, então, a expressão regular `([a-zA-Z])([a-zA-Z]|[0-9]|"_")*`

1.1.7 Declaração e atribuição

A declaração possui a forma `tipo identificador = valor`. A atribuição inicial é obrigatória, portanto a BNF associada à declaração é `type ID "=" expr`. A atribuição possui a forma `identificador = valor`, portanto a BNF associada é `ID "=" expr`

1.2 Operadores

1.2.1 `+`

Soma números e concatena strings. Funciona apenas como operador binário, não sendo possível escrever `+5 - 3`, por exemplo.

1.2.2 `-`

Subtrai números e funciona também como operador unário, sendo possível escrever `-5 - 3`.

1.2.3 `*`, `/`

Multiplicação e divisão entre `integers` e `numbers`.

1.2.4 `mod`

Retorna o resto de uma operação de divisão. Funciona também para o tipo `number`, não se restringindo aos inteiros.

1.2.5 `>`, `<`, `>=`, `<=`, `==` e `not=`

Operadores relacionais

1.3 Comentários

Os comentários são definidos semelhantemente a *C++* e *Java*, ou seja, `//` é usado para comentários de linha e `/* */` é usado para comentários multilinha. Visto que identificar comentários pelo analisador léxico é não-trivial, foram utilizadas as expressões regulares deste site: `"//".*` para comentários de linha e `[/][*][^*]*[*]+([^*/][^*]*[*]+)*[/]` para comentários multilinha. Adicionalmente, a detecção de comentários não terminados é feita através da expressão `[/][*]`.

1.4 Estruturas de controle

1.4.1 if

Executa um bloco de instruções caso a expressão especificada após a palavra *if* seja avaliada como verdadeira. A BNF associada é **`"if" expr {stmt} {"elseif" expr {stmt}} ["else" expr {stmt}] "end"`**

1.4.2 repeat n times

Repete um bloco de instruções n vezes, sendo n qualquer número, inclusive negativo ou zero (que nada executam)

1.4.3 repeat while

Repete um bloco de instruções enquanto uma determinada condição especificada depois da palavra **`while`** for avaliada como verdadeira.

1.4.4 repeat until

Repete um bloco de instruções enquanto uma determinada condição especificada depois da palavra **`while`** for avaliada como falsa.

Desta forma, a BNF para as estruturas de repetição é **`"repeat" (("while" | "until") expr | NUMBER times) "end"`**

1.5 Actions

Um bloco de instruções que, após ser definido, pode ser chamado em qualquer parte do programa. Recebe zero ou mais parâmetros e opcionalmente retorna um valor de um tipo especificado na definição da ação. Temos a BNF **`"action" ID ["(" args_def ")"] ["returns" type] stmt "end"`**

1.6 Classes

De acordo com a documentação, uma classe representa uma coleção de dados e ações sobre esses dados. Para definir uma nova classe, utiliza-se **`class identificador end`**. Apenas declarações de variáveis e ações podem aparecer dentro desse bloco.

1.6.1 Herança

Opcionalmente, uma classe pode herdar atributos e ações de outras classes. Isso é feito escrevendo **`class identificador is identificadores`**, em que **`identificadores`** pode conter uma ou mais classes separadas por vírgula.

Assim, a BNF para declaração de classes é **"class" ID ["is" ID {"," ID}] {declaration | action} "end"**

1.7 Classes genéricas

São definidas como uma classe comum, porém utilizando a forma **class Identificador<Type>** e instanciadas especificando o tipo desejado: **Identificador_da_classe<tipo> identificador**

1.8 Autoboxing

Autoboxing é o nome dado à conversão automática de tipos primitivos realizada ao utilizar classes genéricas.

1.9 Tratamento de erros

Para blocos **try-catch-finally** comumente encontrados em outras linguagens, são utilizadas as palavras **check-detect-always**.

1.10 Estrutura de um programa

É necessário colocar chamadas de ações dentro de um ação especial **Main**, opcionalmente localizada dentro de uma classe **Main**. Fora da classe principal, é possível declarar variáveis, realizar atribuições, definir estruturas de controle e classes e realizar detecção de erros.

1.11 BNF completa

```
type            ::= "integer" | "number" | "boolean" | "text"

un_op           ::= "-" | "not"
bin_op          ::= "+" | "-" | "*" | "/" | ">" | "<" | ">=" | "<=" | "=" | "not="
cast_expr       ::= "cast" "(" type_expr "," ID ")"
expr            ::= expr bin_op expr | un_op expr | "false" | "true"
                    | NUMBER | NAME | cast_expr

lib             ::= ID {"." ID}
args            ::= ID {"," ID}
args_def        ::= type ID {"," type ID}
declaration     ::= type ID "=" expr
action          ::= "action" ID ["(" args_def ")"] ["returns" type] {stmt} "end"
action_call     ::= [ID ":"] ID "(" {args} ")"
class_decl      ::= "class" ID ["is" ID {"," ID}] {declaration | action} "end"

assign_stmt     ::= ID "=" expr
repeat_stmt     ::= "repeat" (("while" | "until") expr | NUMBER times) "end"
if_stmt         ::= "if" expr {stmt} {"elseif" expr {stmt}} ["else" expr {stmt}] "end"
use_stmt        ::= "use" lib
check_stmt      ::= "check" {stmt} "end"
```

```
detect_stmt     ::= "detect" ID "is" ID {stmt} "end"
always_stmt     ::= "always" {stmt} "end"

stmt            ::= assign_stmt | repeat_stmt | if_stmt | class_decl | use_stmt
                  | check_stmt | detect_stmt | always_stmt | declaration

program         ::= {stmt}
```