Universidade Federal de Alagoas (UFAL)
Instituto de Computação (IC)

COMPILADORES

Randy Ambrósio Quindai João
Derecky Costa da Fonseca Andrade

Especificação: Analisador Léxico

Título: Especificação para definição da linguagem RD

Professor: Alcino Dall Igna

9 de abril de 2019

1 Estrutura geral de um programa

A linguagem RD é uma linguagem de programação procedural, projetada para ser analisada em passo único, admite coerção implícita de alguns tipos compatíveis, as palavras reservadas são em inglês, inspiradas na linguagem Pascal. É sensível à caixa, fortemente tipada, podem ser usados os tipos de dados primitivos, estática, ou seja, não há tratamento para erros de detecção de tipo. O código dessa linguagem está disponível no seguinte endereço:

https://github.com/quindai/compilador

Um programa RD inicia com a palavra `pgm`, e termina com a palavra `end_pgm`. O bloco de instruções principal é delimitado pelas palavras `main` seguida de um par de chaves {}. A definição de uma função começa com a palavra `func` seguida do nome da função, parênteses e seus parâmetros, o corpo da função é delimitado por chaves {}. As variáveis devem ser declaradas na área designada, a partir da linha após a palavra `pgm`, dentro de blocos de instruções ou de funções. Variáveis globais são declaradas e inicializadas quando o programa se inicia, variáveis locais são inicializadas a cada vez que o bloco de instrução for chamado.

Estrutura geral de um programa RD:

```
1  pgm
2      <variaveis>
3      <funcoes>
4      main{
5          <instrucoes>;
6      }
7  end_pgm
```

Um programa RD deve estar conforme aos seguintes itens:

- Uma função é iniciada pela palavra reservada `func`, com escopo delimitado por abertura e fechamento de chaves.
- A rotina principal `main` não é uma função, todavia, com escopo delimitado por abertura e fechamento de chaves.

- A rotina `main` é definida após o término das definições das funções

2 Nomes

A linguagem RD não é sensível à caixa, ou seja, não há distinção de maiúscula e minúscula entre palavras reservadas, os nomes devem ter até um tamanho de até 25 caracteres.

Na linguagem RD um nome representa uma palavra reservada ou identificador, a linguagem RD não possui operador ternário.

Palavra reservada são palavras com um significado na linguagem de programação, não podem ser modificadas, usadas como identificadores ou redefinidas. Expressão regular: são as próprias palavras entre aspas.

pgm	end_pgm	main	step	or
func	array	int	if	not
real	string	char	else	and
bool	while	to	switch	mod
repeat	from	case	default	div

Tabela 1: Palavras Reservadas da Linguagem RD

Identificador nomes dos símbolos definidos pelo programador, podem ser modificados e reusados, sujeitos às regras de escopo da linguagem. É caracterizado por qualquer palavra iniciada por uma letra, seguida de letras e números, espaços em branco não podem ser usados, palavras reservadas não podem ser usadas como identificadores. Nenhum operador ou símbolo especial é permitido.

Expressão regular: `[A-Za-z][A-Za-z0-9]*`

Exemplo	Validação
aba	Válido
AbA	Válido
ANDA	Válido
1aba	Não válido
_aba	Não válido
ds:ds	Não válido
or	Não válido
OR	Não válido

Tabela 2: Identificadores válidos e não válidos

Símbolos Especiais são caracteres com significado na linguagem: `[]{}(),;`

[]	usados como referência de elementos de array
()	usados para delimitar os parâmetros de uma função e ordem na precedência de operações
{}	usado para agrupar blocos de instruções
,	usada para separar variavéis ou parâmetros de função
;	terminador de instrução

Tabela 3: Símbolos especiais

Operadores são símbolos que desencadeiam uma ação, podem ser unários ou binários.

**	Unário	~	Unário	>=	Binário
-	Binário	*	Binário	>	Binário
=	Binário	==	Binário	<=	Binário
+	Binário	/	Binário	<	Binário
<>	Binário				

Tabela 4: Operadores suportados

3 Tipos e Estruturas de dados

A linguagem RD suporta vários tipos primitivos e constantes referentes aos mesmos. Os tipos que a linguagem RD suporta são: **int, real, char, string e bool**. Constantes são como variavéis, a única diferença é que o seu valor não pode ser modificado pelo programa uma vez definido.

3.1 Forma de declaração

```
<tipo> <identificador1>,..., <identificadorN>;
<tipo> func <identificador>(<parametros>) {}
<tipo> <identificador>[<tamanho>];
```

3.2 Tipos de dados primitivos

Os tipos primitivos que a linguagem RD suporta são: **int, real, char, string e bool**.

3.2.1 Constantes literais dos tipos

Constante literal ou simplesmente literal, é um valor terminal, número, caractere ou string que poderá estar associado a uma variável ou constante simbólica, geralmente usado como: argumento de uma função; operador numa operação aritmética ou lógica. Um literal sempre representa o mesmo valor, são valores colocados diretamente no código, como o número 5, o caractere 'R' ou a string "Olá Mundo".

Literais numéricos podem ser representados numa variedade de formatos (decimal, hexadecimal, binário, ponto flutuante, octal, etc). Essa versão da linguagem RD não dá suporte aos inteiros Hexadecimais, octais e binários.

3.2.2 Inteiro

Decimal base (10).

- Não pode começar com zero, exceto o caso que seja o próprio zero.
- Não pode conter o ponto decimal.
- Não pode conter vírgulas ou espaços.
- Deve conter apenas dígitos 0-9
- Pode ser precedido pelo unário negativo "~"
- Expressão regular: `[0-9]+`
- Declaração: `int meuinteiro;`

- Exemplos de decimais inteiros válidos: `0 5 127  1002 65535`
- Exemplos de decimais inteiros inválidos: `32,76 1.2 1 27 032 3A`

3.2.3 Ponto Flutuante

Literais de Ponto Flutuante podem ser representados em vários formatos para expressar diferentes variações. O qualificador literal `"f"` força o compilador a tratar o valor como ponto flutuante, precisa ser inserido pelo programador explicitamente.

- Não pode começar com zero, a menos que o zero seja seguido de um ponto decimal.
- Pode usar a notação `"e"` para expressar valores exponenciais (e$\pm$n=10^n)
- Pode conter um ponto decimal
- Não pode conter vírgulas ou espaços
- Deve conter dígitos 0-9
- Pode ser precedido pelo unário negativo `"`$\sim$`"`
- É permitido o qualificador literal `"f"`, forçando o compilador a tratá-lo como real
- Expressão regular: `'f'?[:digit:]+'.'[:digit:]{+}([E|e][+|-]?[:digit:]+)?`
- Declaração: **`real`** `meureal;`
- Exemplos de pontos flutuantes válidos: `2.21e-5 10.22 48e+8 0.5 f10`
- Exemplos de pontos flutuantes inválidos: `02.42 f22 0x5eA`

3.2.4 Caractere

- Deve estar entre apóstrofo (aspas simples)
- Pode conter qualquer caractere imprimível
- Expressão regular: `'[^']'`
- Exemplos de caracteres válidos: `'r', 'R', '\n', '@', '2', ''(espaço)`
- Exemplos de caracteres inválidos: `'me', '''`

3.2.5 String

- Deve estar entre aspas duplas
- Aceita qualquer conjunto de caracteres entre aspas duplas
- Deve começar e terminar na mesma linha
- Expressão regular: `"[^"]*"`
- Declaração: **`string`** `meustring;`
- Exemplos de strings válidos: `"MM", "Nasa", "PC", "A", "sew121@[]"`
- Exemplos de strings inválidos: `2"w", "Ola, "¨¨`

3.2.6 Lógico

É o tipo booleano, com dois únicos possivéis valores, **`true, false`**.

- Declaração: **`bool`** `meubooleano;`

3.2.7 Operações suportadas

Tipo	Operação suportada
Inteiro	atribuição, aritmética, relacional
Ponto Flutuante	atribuição, aritmética, relacional
String	atribuição, relacional, concatenação
Caractere	atribuição, relacional
Lógico	atribuição, relacional, lógico

Tabela 5: Todos os tipos suportam apenas as operações descritas nessa tabela

3.3 Cadeias de caracteres

A palavra reservada **array** permite declarar uma cadeia de caracteres, onde seus literais são um conjunto de caracteres com limitação de tamanho mínimo 0, são delimitados por aspas duplas.

- Declaração: **`array`** `meuarray;`

3.4 Arranjos unidimensionais

Arranjos são variáveis que podem armazenar muitos valores do mesmo tipo, os valores individuais, chamados elementos, são armazenados sequencialmente e são identificados pelo arranjo unicamente por um índice.

- Pode conter qualquer número de elementos
- Elementos têm que ser do mesmo tipo
- Os índices têm que ser do tipo inteiro
- O índice do primeiro elemento é zero
- Os índices não podem ser valores inteiros negativos
- Quando passados como parâmetros de função não se explicita o tamanho do arranjo
- Para variavéis o tamanho do arranjo tem que ser explicitado na sua declaração
- Declaração: **`<tipo>`** `meuarray[<tamanho>];`
- Exemplos:

```
int meuint[12];
real meureal[8];
bool meubool[112];
```

3.5 Equivalência de tipos

A linguagem RD é estaticamente tipada, toda a verificação de compatibilidade de tipos será feita estaticamente. Não admite constante com nome, apenas constantes literais dos tipos são admissivéis.

- Os tipos primitivos usam equivalência de nomes
- Os arranjos são equivalentes de forma estrutural

3.5.1 Coerções admitidas

As seguintes coerções são válidas quando as variavéis são inicializadas, a tentativa de atribuir qualquer valor de um tipo não suportado resultará em erro:

- char para int
- int para char
- int para real
- char para string
- Exemplo:

```
int meuint = 'v';
char meuchar = 22;
real meureal = 10;
string meureal = 'h';
```

3.5.2 Conversão de tipo explícita (cast)

- char para int
- int para char
- int para real (perde-se a parte fracionária)
- real para int
- char para string
- Exemplo:

```
int meuint = (int)'v';
char meuchar = (char)22;
real meureal = (real)10;
int meuint = (int)10.2;
string meureal = (string)'h';
```

4 Atribuição e expressões

Atribuição é uma instrução feita com operador "=". Atribui o valor à direita à variável à esquerda do mesmo.

4.1 Expressões aritméticas, relacionais e lógicas

Lista exaustiva dos operadores.

- Aritméticos

+	Adição
-	Subtração
*	Multiplicação
/	Divisão
**	Exponencial
$\sim$	Unário negativo
div	Divisão inteira
mod	Resto de divisão

Tabela 6: Operadores aritméticos

- Relacional

$>$	Maior que
$<$	Menor que
$==$	Igual a
$<>$	Diferente de
$>=$	Maior ou igual
$<=$	Menor ou igual

Tabela 7: Operadores relacionais

- Lógicos

and	Conjunção
or	Disjunção
not	Negação

Tabela 8: Operadores lógicos

4.2 Precedência e Associatividade

Na tabela a seguir os operadores agrupados na mesma seção têm a mesma precedência, as subsequentes secções têm precedência mais baixa, a associatividade também pode ser observada. Quando expressões são formadas por múltiplos operadores, a precedência determina a ordem de avaliação, quando dois operadores possuem a mesma precedência, a associatividade determina a ordem de avaliação.

Operador	Descrição	Associatividade
`()`	Expressão em parêntesis	Dentro para fora
`[]`	Descritor de tamanho de arranjo	
$\sim$	Unário negativo	Direita para esquerda
`not`	NOT lógico	
`**`	Exponencial	
`* /`	Multiplicação, divisão, módulo, divisão	Esquerda para direita
`mod`	inteira	
`div`		
`+ -`	Soma, subtração	Esquerda para direita
`< <=`	Menor que, Menor que ou igual	Esquerda para direita
`> >=`	Maior que, Maior que ou igual	
`== <>`	Igual, Não igual	Esquerda para a direita
`and`	AND lógico	Esquerda para a direita
`or`	OU lógico	

Tabela 9: Precedência e associatividade de operadores

5 Sintaxe e exemplo de estruturas de controle

Esses comandos oferecem instruções para tomada de decisão. Condição representa um valor lógico, true ou false.

5.1 Estrutura condicional de uma e duas vias

- `if (<condição>) <instruções>`
- `if (<condição>) <instruções> else <instruções>`
- `switch <variável> case <condição>: <instruções>; default: <instruçõe>`

5.1.1 Semântica

Para a instrução `if`: se condição for verdadeira executa bloco de instruções, caso contrário o bloco de instruções associado ao `else` subsequente ao `if` será executado, o bloco de instruções só será executado se a condição não for verdadeira.

Para a instrução `switch`: o valor da `variável` é avaliado em todas as condições `case`, se nenhuma das condições `case` for satisfeita, o bloco de instruções associado ao `default` é executado. O comando termina a sua execução quando encontra um `case` com condição verdadeira.

5.2 Estrutura iterativa com controle lógico

Esse tipo de comando permite a execução de instruções até que uma dada condição seja satisfeita.

- `while (<condição_bool>) <instruções>`

5.2.1 Semântica

Se condição booleana for verdadeira (true), o conjunto de instruções é executado, esse processo será repetido até que a condição booleana seja falsa.

5.3 Estrutura iterativa controlada por contador com passo igual a um caso omitido

- **repeat** <identificador> **from** <expressão1>
  to <expressão2> [**step** <expressão3>]? <instruções>

5.3.1 Semântica

Esse comando vai executar um conjunto de instruções enquanto o valor da `expressão1` for igual ou menor ao valor da `expressão2`, onde *expressãoN* é uma expressão que retorna um valor resultante de uma operação aritmética, `expressão2` é incrementada em uma unidade ou num valor definido na `expressão3`. O valor da *expressãoN* é pré-avaliado e armazenado numa variável temporária definida pelo compilador.

6 Subprogramas

6.1 Funções

São segmentos de programas com a finalidade de resolver uma tarefa específica bem definida. Toda a função retorna um valor de um tipo, todo o valor passado para uma função é copiado para um escopo local. Os parâmetros de uma função são declarados como se declaram as variáveis, separados por vírgula e delimitados por parênteses. Os valores passados nos parâmetros não afetam a variável que continha o valor inicialmente.

A declaração de uma função define um identificador e o associa a um bloco de código, o bloco irá retornar um valor que será passado a esse identificador.

- [<tipo>] func <identificador> (ϵ | <parâmetros>) {
  <variáveis>; <instruções>; return <valor>}

- Exemplo:

```
int func soma(int a, int b) {
int retorno;
retorno = a + b
return retorno;
}
real func maior(real a, real b) {
if (a > b)  return a;
return b;
}
```

7 Comentários

Comentários são linhas ou blocos de texto usados para documentar a funcionalidade de um programa e explicar como um programa funciona, têm a finalidade de beneficiar o programador. Comentários são ignorados pelo compilador. RD suporta apenas comentário de linha, todos os caracteres da linha serão ignorados após o símbolo //.

- Exemplo: `// isso é um comentário`

8 Escopo

Como RD é uma linguagem analisada em passo único, os identificadores (variáveis e funções) não declarados antes da sua utilização serão considerados identificadores não declarados, mesmo que sejam declarados em algum ponto do programa após a instrução que tentar usá-lo.

8.1 Analisador Léxico

Na linguagem de programação RD há 5 categorias de tokens: palavras reservadas, identificadores, operadores, constantes, separadores e símbolos especiais.

8.1.1 Operadores

Símbolos para as operações definidas na linguagem.

8.1.2 Separadores

São espaços em branco, ponto, vírgulas e ponto-e-vírgula.

8.1.3 Constantes

Denotam um valor, numérico ou não, colocado no código fonte (uma constante literal).

8.2 Expressões Regulares

- O símbolo ϵ significa produção vazia, palavra vazia ou ausência de tokens.
- O símbolo — separa as alternativas das produções, pode-se ler como OU.
- Espaço branco é uma sequência não vazia de espaços, novas linhas e tabs.

letra = [a-zA-Z]

digito = [:digit:]

palavra_chave = `'if'| 'else'| 'for'| 'main'| 'func'| 'while'| 'repeat'| 'int'| 'array'| 'real'| 'bool'| 'string'| 'char'| 'pgm'| 'end_pgm'| 'true'| 'false'| 'return'| 'and'| 'or'| 'not'| 'div'| 'mod'| 'break'| 'from'| 'to'`

op_aritmetico = `'+'| '-'| '*'| '**'| '/'| ' '`

op_relacional = `'<'| '>'| '=='| '<='| '>='| '<>'`

separadores = `'.'| ','| ';'| ''`

simbolos = `'{'| '}'| '('| ')'| '['| ']'| op_aritmetico | op_relacional | separadores`

identificador = `[letra][:alnum:]*`

espaco_branco = `[:blank:]`

const_num = `'0'|([digito]{-}['0'])[digito]*`

signal_num = `(ϵ|['+'|'~'][const_num])`

float = `signal_num('.'const_num)| const_num([E|e][+-]?digito+)?`

8.3 Tokens

A lista a seguir lista todos os tokens com as suas respectivas categorias simbólicas:

Num.	Token	Categoria Simbólica	Expressão Regular
0	pgm	PGM	`(i:"pgm")`
1	int	RD_INT	`(i:"int")`
2	real	RD_REAL	`(i:"real")`
3	string	RD_STRING	`(i:"string")`
4	char	RD_CHAR	`(i:"char")`
5	bool	RD_BOOL	`(i:"bool")`
6	array	RD_ARRAY	`(i:"array")`
7	if	IF	`(i:"if")`
8	else	ELSE	`(i:"else")`
9	while	WHILE	`(i:"while")`
10	return	RD_RETURN	`(i:"return")`
11	from	FROM	`(i:"from")`
12	repeat	REPEAT	`(i:"repeat")`
13	main	MAIN	`(i:"main")`
14	end_pgm	END_PGM	`(i:"end_pgm")`
15	to	TO	`(i:"to")`
16	true	TRUE	`(i:"true")`
17	false	FALSE	`(i:"false")`
18	print	PRINT	`(i:"print")`
19	func	FUNC	`(i:"func")`
20	step	STEP	`(i:"step")`
21	rd_error	RD_ERROR	
22	identifier	IDENTIFIER	`[letra][:alnum:]*`
23	lit_int	LIT_INT	`[0-9]+`
24	lit_char	LIT_CHAR	`'[^']'`
25	lit_string	LIT_STRING	`"[^"]*"`
26	lit_bool	LIT_BOOL	
27	lit_real	LIT_REAL	`'f'?[:digit:]+'.'[:digit:]{+}([E|e][+|-]?[:digit:]+)?`
28	==	EQ	`'=='`
29	∼	UNARY	`"∼"`
30	*	MULT	`"*"`
31	**	POW	`"**"`
32	+	PLUS	`"+"`
33	-	MINUS	`-"`
34	mod	MOD	`(i:"mod")`
35	div	INTDIV	`(i:"div")`
36	or	OR	`(i:"or")`
37	not	NOT	`(i:"not")`
38	and	AND	`(i:"and")`
39	<>	NE	`"<>"`
40	<	LT	`"<"`
41	<=	LE	`"<="`
42	>	GT	`">"`
43	>=	GE	`">="`
44	//	COMMENT	`"//"`
45	=	ASSIGN	`"="`

46	]	SRBRAC	"]"
47	[	SLBRAC	"["
48	/	DIVIDE	"/"
49	)	RPAREN	")"
50	(	LPAREN	"("
51	}	RBRAC	"}"
52	{	LBRAC	"{"
53	:	COLON	":"
54	;	SEMICOLON	";"
55	,	COMA	","
56	"	DOUBLE_QUOTES	'"'

9 Programas Exemplo

Alô mundo

```
pgm
main{
print("Alô mundo");
}
end_pgm
```

Série de Fibonacci

```
pgm
int valor;
func fibo(int n){
int prev, atual, temp, i;
prev = 0;
i = 0;
atual = 1;
repeat i from 0 to n{
if(i==0){print(i +",");}
else{
if(i < 2){print(atual +",");}
else{
temp = atual;
atual = atual + prev;
prev = temp;
print(atual +",");
}}
}
main{
fibo(10)
}
end_pgm
```

Shell Sort

```
pgm //Shell sort
int vet[10];
int n;
int func shellsort(int v[], int tam){
int i, j, h;
int gap = 1;
while(gap < tam){
gap = 3*gap+1;
}
while(gap > 1) {
gap = gap / 3;
repeat i from gap to size{
h = v[i];
j = i;
while(j >= gap and h < v[j-gap]){
vet[j] = v[j-gap];
j = j-gap;
}
v[j] = value;
}
}
return v;
}
main{
vet = shellsort(vet, 10);
}
end_pgm
```