CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS

DANIEL DE PAULA BRAGA LOPES
GUILHERME FERNANDES MARCHEZINI
LAURO CÉSAR JACQUES SANTOS

Relatório Gerador de Código

Belo Horizonte
2017

1. Forma de Utilização

A execução do compilador é feita via terminal do Linux utilizando o `.jar` que está localizado na pasta dist passando o nome do `.jar` e o arquivo de entrada que será compilado, da seguinte forma:

```
java -jar Compilador.jar <nome_aquivo_entrada> (<nome_aquivo_saida>)
```

Abaixo um exemplo de como compilar o primeiro código de teste disponibilizado na especificação do trabalho:

```
java -jar Compilador.jar ../test/teste1.tst ../test/teste1.vm
```

2. Modificações

A gramática original da linguagem apresenta prefixos comuns e recursão à esquerda, elementos que impossibilitam a implementação de um parser LL(1) para a mesma. A seguir a gramática modificada, solucionando tais problemas e reescrita no formato BNF:

```
program'            ::= program "$"
program             ::= init decl-stmt-list stop
decl-stmt-list      ::= "id" assign-or-decl | stmt-no-assign ";" stmt-list-tail
assign-or-decl      ::= ":=" simple-expr ";" stmt-list-tail
                        | ident-list-tail is type ";" decl-stmt-list-tail
stmt-no-assign      ::= if-stmt | do-stmt | read-stmt | write-stmt
decl-stmt-list-tail ::= decl-stmt-list | λ
ident-list-tail     ::= "," "id" ident-list-tail | λ
type                ::= integer | string
stmt-list           ::= stmt ";" stmt-list-tail
stmt-list-tail      ::= stmt ";" stmt-list-tail | λ
stmt                ::= assign-stmt | if-stmt | do-stmt
                        | read-stmt | write-stmt
assign-stmt         ::= "id" ":=" simple_expr
if-stmt             ::= if "(" condition ")" begin stmt-list end if-suffix
if-suffix           ::= else begin stmt-list end | λ
condition           ::= expression
do-stmt             ::= do stmt-list do-suffix
do-suffix           ::= while "(" condition ")"
read-stmt           ::= read "(" "id" ")"
write-stmt          ::= write "(" writable ")"
writable            ::= simple-expr
expression          ::= simple-expr expression-suffix
expression-suffix   ::= relop simple-expr | λ
simple-expr         ::= term simple-expr-tail
simple-expr-tail    ::= addop term simple-expr-tail | λ
term                ::= factor-a term-tail
term-tail           ::= mulop factor-a term-tail | λ
factor-a            ::= factor | not factor | "-" factor
factor              ::= "id" | constant | "(" expression ")"
relop               ::= "=" | ">" | ">=" | "<" | "<=" | "<>"
addop               ::= "+" | "-" | or
mulop               ::= "*" | "/" | and
constant            ::= "num" | "literal"
```

Obs.: Algumas produções da gramática original foram substituídas por tokens.

Para a implementação do parser LL(1), foram encontrados os conjuntos FIRST e FOLLOW de todos os símbolos não terminais da gramática:

Símbolo	First	Follow
program	init	$
decl-stmt-list	id, if, do, read, write	stop
assign-or-decl	:=, ",", is	stop
stmt-no-assign	if, do, read, write	;
decl-stmt-list-tail	id, if, do, read, write, λ	stop
ident-list-tail	",", λ	is
type	integer, string	;
stmt-list	id, if, do, read, write	end, while
stmt-list-tail	id, if, do, read, write, λ	stop, end, while
stmt	id, if, do, read, write	;
assign-stmt	id	;
if-stmt	if	;
if-suffix	else, λ	;
condition	id, num, literal, (, not, -	)
do-stmt	do	;
do-suffix	while	;
read-stmt	read	;
write-stmt	write	;
writable	id, num, literal, (, not, -	)
expression	id, num, literal, (, not, -	)
expression-suffix	>, =, >=, <, <=, <>, λ	)
simple-expr	id, num, literal, (, not, -	;,), >, =, >=, <, <=, <>
simple-expr-tail	or, +, -, λ	;,), >, =, >=, <, <=, <>
term	id, num, literal, (, not, -	or, +, -, ;,), >, =, >=, <, <=, <>
term-tail	*, /, and, λ	or, +, -, ;,), >, =, >=, <, <=, <>
factor-a	id, num, literal, (, not, -	*, /, and, or, +, -, ;,), >, =, >=, <, <=, <>
factor	id, num, literal, (	*, /, and, or, +, -, ;,), >, =, >=, <, <=, <>
relop	>, =, >=, <, <=, <>	id, num, (, not, -
addop	or, +, -	id, num, (, not, -
mulop	*, /, and	id, num, (, not, -
constant	num, literal	*, /, and, or, +, -, ;,), >, =, >=, <, <=, <>

3. Implementação

Abaixo uma breve explicação das classes existentes no compilador:

3.1 Lexer

Classe que implementa o analisador léxico. Seu construtor insere as palavras reservadas na tabela de símbolos. Possui um método **scan** que devolve um Token.

3.2 LexicalException

Classe para imprimir na tela o motivo de ocorrer uma determinada exceção. Existem três casos:

- Token inválido: É passado um token inválido ou não esperado;
- Fim de arquivo inesperado: O arquivo termina quando ainda deveria possuir alguma informação;
- Default: Ocorre quando é um erro diferente dos dois anteriores.

3.3 Num

Classe para representar um Token número.

3.4 Tag

Classe que define as constantes para os tokens.

3.5 Token

Representa um Token genérico. Contém a constante que representa o Token.

3.6 Word

Representa um token de palavras reservadas, identificadores e tokens compostos, tais como != e &&.

3.7 Syntaxer

Essa classe implementa completamente o parser LL(1) com todas os módulos necessários, como por exemplo o ‘eat’ e ‘advance’, sendo responsável também pela recuperação de erros, que foi implementada usando a heurística dos follows.

3.8 SyntaticException

Classe para imprimir na tela os erros de sintaxe encontrados no programa fonte.

3.9 Command

Indica se um comando teve erro semântico ou não através do Type.

3.10 Operation

Define um valor para cada uma das operações possíveis. Ex: ADD=6, GTE=2.

3.11 SemanticException

A classe possui método que imprime erro semântico quando ele ocorrer.

3.12 Type

Classe possui variáveis estáticas que definem constantes para o Tipo, sendo eles Error, Null, Integer, String e Boolean. Também possui métodos que atribuem o tipo e a largura.

3.13 Expression

Dada uma expressão ou operação envolvendo variáveis ou constantes, atribui o tipo resultante certo ou erro para a expressão.

3.13 DeclarationCommand

Mantém uma lista de identificadores declarados para no momento que o tipo for informado atualizar a tabela de símbolos atribuindo o tipo aos identificadores.

3.13 Code

Mantém a lista de instruções que serão geradas ao final da compilação, permitindo que essas sejam “remendadas” e que labels sejam adicionadas quando necessário.

3.13 Label

Classe responsável pela geração de labels sequenciais que nunca se repetem. As labels iniciam em L0 e se estendem até LN, sendo o N o número de instruções em que são destino de desvios.

4. Testes

A seguir os testes e seus respectivos resultados:

4.1 Teste 1

Código:

```
init
    a, b, c, result is integer;

    read(a);
    read(c);
    b := 10;
    result := (a * c) / (b + 5 - 345);
    write(result);
stop
```

Código objeto:

```
START
PUSHN 4
READ
ATOI
STOREG 3
READ
ATOI
STOREG 1
PUSHI 10
STOREG 0
PUSHG 3
PUSHG 1
MUL
PUSHG 0
PUSHI 5
ADD
PUSHI 345
SUB
DIV
STOREG 2
PUSHG 2
WRITEI
STOP
```

Execução:

```
VM version 1.6
read =>
10
read =>
60
-1
```

4.2 Teste 2

Código:

```
init
    a, valor, b is integer;

    read(a);
    b := a * a;
    write(b);
    b := b + a / 2 * (a + 5);
    Write(b);
stop
```

Código objeto:

```
START
PUSHN 3
READ
ATOI
STOREG 2
PUSHG 2
PUSHG 2
MUL
STOREG 1
PUSHG 1
WRITEI
PUSHG 1
PUSHG 2
PUSHI 2
DIV
PUSHG 2
PUSHI 5
ADD
MUL
ADD
STOREG 1
PUSHG 1
WRITEI
STOP
```

Execução:

```
VM version 1.6
read =>
4
16
34
```

4.3 Teste 3

Código:

```
{ Programa de Teste
Calculo de idade }
init
	cont_, qtd is integer;
	media, idade, soma, altura is integer;

	cont_ := 5;
	soma := 0;
	do
		write("Altura: ");
		read (altura);
		soma := soma + altura;
		cont_ := cont_ - 1;
	while (cont_ > 0);

	write("Media: ");
	write(soma / qtd);
stop
```

Código objeto:

```
START
PUSHN 2
PUSHN 4
PUSHI 5
STOREG 1
PUSHI 0
STOREG 3
L0: PUSHS "Altura: "
WRITES
READ
ATOI
STOREG 4
PUSHG 3
PUSHG 4
ADD
STOREG 3
PUSHG 1
PUSHI 1
SUB
STOREG 1
L1: PUSHG 1
PUSHI 0
SUP
JZ L2
JUMP L0
JUMP L1
L2: PUSHS "Media: "
WRITES
PUSHG 3
PUSHG 0
DIV
WRITEI
STOP
```

Execução:

```
VM version 1.6
Altura:
read =>
1
Altura:
read =>
2
Altura:
read =>
3
Altura:
read =>
4
Altura:
read =>
5
Media:
VM error: Division By Zero
```

4.4 Teste 4

Código:

```
init
    i, j, k, total, soma, a is integer;

    read(I);
    k := i * (5 - i * 50 / 10);
    j := i * 10;
    k := i * j / k;
    k := 4 + a;
    write(i);
    write(j);
    write(k);
stop
```

Código objeto:

```
START
PUSHN 6
READ
ATOI
STOREG 5
PUSHG 5
PUSHI 5
PUSHG 5
PUSHI 50
MUL
PUSHI 10
DIV
SUB
MUL
STOREG 1
PUSHG 5
PUSHI 10
MUL
STOREG 0
PUSHG 5
PUSHG 0
MUL
PUSHG 1
DIV
STOREG 1
PUSHI 4
PUSHG 4
ADD
STOREG 1
PUSHG 5
WRITEI
PUSHG 0
WRITEI
PUSHG 1
WRITEI
STOP
```

Execução:

```
VM version 1.6
read =>
100
100
1000
4
```

4.5 Teste 5

Código:

```
init
// Programa com if
    k, m, result is integer;
    a, j is string;

    read(j);
    read(k);

    if (j = "ok")
    begin
        result := k/m
    end
    else
    begin
        result := 0;
        write ("Invalid entry");
    end;

    write(result);
stop
```

Código objeto:

```
START
PUSHN 3
PUSHN 2
READ
STOREG 3
READ
ATOI
STOREG 2
PUSHG 3
PUSHS "ok"
EQUAL
JZ L1
JUMP L0
L0: PUSHG 2
PUSHG 0
DIV
STOREG 1
JUMP L2
L1: PUSHI 0
STOREG 1
PUSHS "Invalid entry"
WRITES
L2: PUSHG 1
WRITEI
STOP
```

Execuções:

```
VM version 1.6
read =>
ok
read =>
10
VM error: Division By Zero

VM version 1.6
read =>
notok
read =>
10
Invalid entry
0
```

4.6 Teste 6

Código:

```
init
    a, b, c, maior is integer;
    read(a);
    read(b);
    read(c);
    maior := 0;
    if ((a > b) and (a > c))
    begin
        maior := a;
    end
    else
    begin
        if (b > c)
        begin
            maior := b;
        end
        else
        begin
            maior := c;
        end;
    end;
    write("Maior idade: ");
    write(maior);
stop
```

Código objeto:

```
START
PUSHN 4
READ
ATOI
STOREG 3
READ
ATOI
STOREG 0
READ
ATOI
STOREG 1
PUSHI 0
STOREG 2
PUSHG 3
PUSHG 0
SUP
JZ L4
JUMP L0
L0: PUSHG 3
PUSHG 1
SUP
JZ L4
JUMP L3
MUL
L3: PUSHG 3
STOREG 2
JUMP L5
L4: PUSHG 0
PUSHG 1
SUP
JZ L2
JUMP L1
L1: PUSHG 0
STOREG 2
JUMP L5
L2: PUSHG 1
STOREG 2
L5: PUSHS "Maior idade: "
WRITES
PUSHG 2
WRITEI
STOP
```

Execuções:

```
VM version 1.6
read => 1
read => 2
read => 3
Maior idade: 3
```

```
VM version 1.6
read => 1
read => 3
read => 2
Maior idade: 3
```

```
VM version 1.6
read => 3
read => 2
read => 1
Maior idade: 3
```

4.7 Teste 7

Código:

```
init
    a is integer;

    read(A);

    DO
       A := A - 2;
    WHiLE (A >= 2);

    iF (a = 0)
    begin
       write(A);
       write(" é par.");
    end
    ELSE
    begin
       write(" é ímpar.");
    end;
stop
```

Código objeto:

```
START
PUSHN 1
READ
ATOI
STOREG 0
L0: PUSHG 0
PUSHI 2
SUB
STOREG 0
L1: PUSHG 0
PUSHI 2
SUPEQ
JZ L4
JUMP L0
JUMP L1
L4: PUSHG 0
PUSHI 0
EQUAL
JZ L3
JUMP L2
L2: PUSHS " é par."
WRITES
JUMP L5
L3: PUSHS " é ímpar."
WRITES
L5: STOP
```

Execuções:

```
VM version 1.6
read =>
15
 é ímpar.

VM version 1.6
read =>
30
 é par.
```

4.8 Teste 8

Código:

```
init
    n is integer;
    anterior, proximo, aux, i is
integer;

    write("Digite a posicao: ");
    read(n);

    if (n = 1)
    begin
        proximo := 0;
    end
    else
    begin
        if (n = 2)
        begin
            proximo := 1;
        end
        else
        begin
            anterior := 1;
            proximo := 1;
            i := 3;
            do
                aux := proximo;
                proximo := anterior +
proximo;
                anterior := aux;
                i := i + 1;
            while (i < n);
        end;
    end;

    write("O termo: ");
    write(proximo);
stop
```

Código objeto:

```
START
PUSHN 1
PUSHN 4
PUSHS "Digite a posicao: "
WRITES
READ
ATOI
STOREG 0
PUSHG 0
PUSHI 1
EQUAL
JZ L5
JUMP L4
L4: PUSHI 0
STOREG 1
JUMP L6
L5: PUSHG 0
PUSHI 2
EQUAL
JZ L3
JUMP L2
L2: PUSHI 1
STOREG 1
JUMP L6
L3: PUSHI 1
STOREG 4
PUSHI 1
STOREG 1
PUSHI 3
STOREG 3
L0: PUSHG 1
STOREG 2
PUSHG 4
PUSHG 1
ADD
STOREG 1
PUSHG 2
STOREG 4
PUSHG 3
PUSHI 1
ADD
STOREG 3
L1: PUSHG 3
PUSHG 0
INF
JZ L6
JUMP L0
JUMP L1
L6: PUSHS "O termo: "
WRITES
PUSHG 1
WRITEI
STOP
```

Execuções:

```
VM version 1.6
Digite a posicao:
read => 1
O termo: 0
```

```
VM version 1.6
Digite a posicao:
read => 2
O termo: 1
```

```
VM version 1.6
Digite a posicao:
read => 15
O termo: 377
```