

Relatório da Versão 3.1 do Compilador

Lógica da Computação - Insper 2024.1

Antônio Amaral Eygdio Martins

1. Introdução

O compilador desenvolvido em Dart é capaz de compilar um código fonte escrito em Lua, com a utilização de variáveis locais e globais, operações aritméticas, operações lógicas, operações de comparação, estruturas condicionais, estruturas de repetição, funções e chamadas de funções.

Em sua versão 3.1 o compilador é capaz de gerar código asm para execução em linux x86-64.

2. Estrutura do Código

2.1. tokenizer.dart

- `tokenizer.dart` : Responsável pela tokenização do código fonte Lua, convertendo-o em uma sequência de tokens que podem ser processados pelo parser.

As diferenças do arquivo na versão 3.0 para a versão 3.1 são:

1. Novos Tokens Adicionados

Foram adicionados os tokens `function` , `return` e `comma` :

```
enum TokenType {  
  integer,  
  string,  
  plus,  
  minus,  
  multiply,  
  divide,  
  eof,  
  openParen,  
  closeParen,  
  identifier,  
  print,  
  equal,  
  or,  
  and,  
  not,  
  greater,  
  less,  
  equalEqual,  
  endToken,  
  ifToken,  
  elseToken,  
  doToken,  
  thenToken,  
  whileToken,  
  read,  
  local,
```

```

    lineBreak,
    concat,
    function,
    RETURN,
    comma,
  }

```

Motivo da adição dos tokens:

- Funções: a inclusão de tokens para funções (`function` , `RETURN`) permite a definição e chamada de funções, suportando recursão.
- Parâmetros: O token `comma` é utilizado para separar parâmetros em definições e chamadas de funções.

2.2. main.dart

1. Nova Classe `FuncTable` :

Esta classe é responsável por armazenar definições de funções e garantir que não haja funções com o mesmo nome.

```

class FuncTable {
  FuncTable._privateConstructor();

  static final FuncTable _instance = FuncTable._privateConstructor();

  static FuncTable get instance => _instance;

  final Map<String, dynamic> _table = {};

  void set({required String key, required Node node}) {
    if (_table[key] == null) {
      _table[key] = node;
    } else {
      throw Exception('Function already defined: $key');
    }
  }

  dynamic get(String key) {
    final value = _table[key];
    if (value == null) {
      throw Exception('Undefined function: $key');
    }
    return value;
  }
}

```

Está não é uma mudança exclusiva da versão 3.1, uma vez que foi implementada na versão 2.4, porém está sendo citada neste relatório uma vez que a versão 3.0 não tinha capacidade de armazenar funções.

2. Atualização em `SymbolTable` :

```

class SymbolTable {
    SymbolTable._privateConstructor();

    static final SymbolTable _instance = SymbolTable._privateConstructor();

    static SymbolTable get instance => _instance;

    static SymbolTable getNewInstance() {
        return SymbolTable._privateConstructor();
    }

    final Map<String, Map<String, dynamic>> _table = {};

    int _offset = 4;

    void set(
        {required String key,
         required dynamic value,
         required dynamic type,
         required bool isLocal}) {
        if (!isLocal) {
            if (_table[key] != null) {
                final auxOffset = _table[key]['offset'];
                _table[key] = {'value': value, 'type': type, 'offset': auxOffset};
            } else {
                throw Exception('Variable already defined: $key');
            }
        } else if (_table[key] == null) {
            _table[key] = {'value': value, 'type': type, 'offset': _offset};
            print('Local variable: $key, offset: $_offset');
            _offset += 4;
        } else {
            throw Exception('Variable already defined: $key (local)');
        }
    }

    void setLocalFunction({
        required String key,
        required dynamic value,
        required dynamic type,
        int additionalOffset = 0,
        int signal = 1,
    }) {
        if (_table[key] == null) {
            _table[key] = {
                'value': value,
                'type': type,
                'offset': (_offset + additionalOffset) * signal
            };
            _offset += 4;
        } else {

```

```

        throw Exception('Function already defined: $key');
    }
}

({dynamic value, String type, int offset}) get(String key) {
    final value = _table[key];
    if (value == null) {
        throw Exception('Undefined variable: $key');
    }
    return (value: value['value'], type: value['type'], offset: _offset);
}

int getOffset(String key) {
    final value = _table[key];
    if (value == null) {
        throw Exception('Undefined variable: $key');
    }
    return value['offset'];
}
}

```

Foram realizadas modificações na `SymbolTable` para possibilitar a criação de funções com variáveis locais, havendo a necessidade de se construir, a nível de stack de memória do computador, um espaço para armazenar variáveis locais e parâmetros de funções.

Cada função do programa em lua terá seu próprio stack frame, onde serão armazenadas as variáveis locais, parâmetros da função e informações de retorno. Este stack frame é criado a partir do uso da função `setLocalFunction` que irá gerenciar o escopo de variáveis alocando elas positivamente ou negativamente na stack de memória.

Adicionalmente é utilizado o parametro `additionalOffset` para alocar variáveis locais em posições específicas da stack de memória, permitindo a alocação de variáveis locais em posições específicas da stack de memória.

3. Atualização no Parser:

Agora o parser têm suporte para funções e retornos.

```

else if (tokenizer.next.type == TokenType.function) {
    tokenizer.selectNext(); // Consume 'function'
    if (tokenizer.next.type != TokenType.identifier) {
        throw FormatException(
            "Expected identifier but found ${tokenizer.next.type}");
    }
    final Token identifier = tokenizer.next;
    final Identifier id = Identifier(identifier.value);
    tokenizer.selectNext();
    if (tokenizer.next.type != TokenType.openParen) {
        throw FormatException("Expected '(' but found ${tokenizer.next.type}");
    }
    tokenizer.selectNext(); // Consume '('
    List<Identifier> parameters = [];
    if (tokenizer.next.type == TokenType.identifier) {

```

```

    final Token parameter = tokenizer.next;
    parameters.add(Identifier(parameter.value));
    tokenizer.selectNext();
    while (tokenizer.next.type == TokenType.comma) {
        tokenizer.selectNext(); // Consume ','
        if (tokenizer.next.type != TokenType.identifier) {
            throw FormatException(
                "Expected identifier but found ${tokenizer.next.type}");
        }
        final Token parameter = tokenizer.next;
        parameters.add(Identifier(parameter.value));
        tokenizer.selectNext();
    }
}

if (tokenizer.next.type != TokenType.closeParen) {
    throw FormatException("Expected ')' but found ${tokenizer.next.type}");
}

tokenizer.selectNext(); // Consume ')'

if (tokenizer.next.type != TokenType.lineBreak) {
    throw FormatException(
        "Expected line break but found ${tokenizer.next.type}");
}

tokenizer.selectNext();

final Node block = this.endBlock();

if (tokenizer.next.type != TokenType.endToken) {
    throw FormatException(
        "Expected 'end' but found ${tokenizer.next.type}");
}

tokenizer.selectNext(); // Consume 'end'

return FuncDecOp(id, parameters, block);
}

```

```

else if (tokenizer.next.type == TokenType.RETURN) {
    tokenizer.selectNext(); // Consume 'return'
    final Node expression = boolExpression();
    return ReturnOp(expression);
}

```

Outras modificações foram realizadas na EBNF e diagrama sintático, porém não serão citadas diretamente neste relatório por serem alteradores totalmente referentes a versão 2.4.

2.3. operands.dart

As maiores modificações foram realizadas neste arquivo, uma vez que agora nenhum dos operadores realizam operações diretamente, apenas retornam o código asm correspondente - todas as operações serão feitas exclusivamente pelo programa em asm.

Um exemplo desta simplificação está na construção da Classe `BinOp` :

```
class BinOp extends Node {
    final Node left;
    final Node right;
    BinOp(this.left, this.right, String op) : super(op);

    @override
    dynamic Evaluate(SymbolTable _table, FuncTable _funcTable) {
        Write write = Write();
        right.Evaluate(_table, _funcTable);
        write.code += "PUSH EAX\n";
        left.Evaluate(_table, _funcTable);
        write.code += "POP EBX\n";

        switch (value) {
            case "TokenType.plus":
                write.code += "ADD EAX, EBX\n";
                break;
            case "TokenType.minus":
                write.code += "SUB EAX, EBX\n";
                break;
            case "TokenType.multiply":
                write.code += "IMUL EAX, EBX\n";
                break;
            case "TokenType.divide":
                write.code += "IDIV EBX\n";
                break;
            case "TokenType.greater":
                write.code += "CMP EAX, EBX\nCALL binop_jg\n";
                break;
            case "TokenType.less":
                write.code += "CMP EAX, EBX\nCALL binop_jl\n";
                break;
            case "TokenType.equalEqual":
                write.code += "CMP EAX, EBX\nCALL binop_je\n";
                break;
            case "TokenType.and":
                write.code += "AND EAX, EBX\n";
                break;
            case "TokenType.or":
                write.code += "OR EAX, EBX\n";
                break;
            default:
                throw Exception('Invalid operator');
        }
    }
}
```

```
}  
}
```

2.3.1. FuncDecOp

Focando agora nas alterações realizadas diretamente na criação e chamada de funções, houve a necessidade de alterar a lógica por trás da criação de funções, ao comparar com a versão 2.4 do compilador, uma vez que a classe `FuncDecOp` não realiza mais apenas a função de armazenar a função na `FuncTable`, como também executa a função realizando o evaluate de seu block.

```
class FuncDecOp extends Node {  
    final Identifier identifier;  
    final List<Identifier> parameters;  
    final Node block;  
    FuncDecOp(this.identifier, this.parameters, this.block) : super(null);  
  
    @override  
    dynamic Evaluate(SymbolTable _table, FuncTable _funcTable) {  
        _funcTable.set(  
            key: identifier.name,  
            node: this,  
        );  
        Write write = Write();  
        write.code += "JMP END_${identifier.name}\n";  
        write.code += "${identifier.name}:\n";  
        write.code += "PUSH EBP\n";  
        write.code += "MOV EBP, ESP\n";  
  
        final localTable = SymbolTable.getNewInstance();  
  
        for (var param in parameters) {  
            localTable.setLocalFunction(  
                key: param.name,  
                value: null,  
                type: null,  
                additionalOffset: 4,  
                signal: -1,  
            );  
        }  
  
        block.Evaluate(localTable, _funcTable);  
  
        write.code += "MOV ESP, EBP\n";  
        write.code += "POP EBP\n";  
        write.code += "RET\n";  
        write.code += "END_${identifier.name}:\n";  
    }  
}
```

Nestá parte do código temos a adição de:

```
write.code += "JMP END_${identifier.name}\n";
```

Para pular a declaração da função e ir direto para o bloco de execução da função. Este código "protege" a declaração da função garantindo que a mesma será executada apenas quando chamada.

```
function fatorial(n)
  if n == 0 then
    return 1
  else
    return n * fatorial(n - 1)
  end
end
```

Sem este código a função `fatorial` seria executada no momento de sua declaração, o que não é o comportamento esperado.

```
write.code += "${identifier.name}:\n";
write.code += "PUSH EBP\n";
write.code += "MOV EBP, ESP\n";
```

Estas linhas de código são responsáveis por criar o stack frame da função, armazenando o valor de `EBP` e movendo o valor de `ESP` para `EBP`.

```
final localTable = SymbolTable.getNewInstance();

for (var param in parameters) {
  localTable.setLocalFunction(
    key: param.name,
    value: null,
    type: null,
    additionalOffset: 4,
    signal: -1,
  );
}
```

Estas linhas de código são responsáveis por criar um novo escopo de variáveis locais para a função, alocando as variáveis locais e parâmetros da função na stack de memória.

```
block.Evaluate(localTable, _funcTable);
```

Esta linha de código é responsável por executar o bloco de código da função, passando o escopo de variáveis locais e a tabela de funções para o bloco.

```
write.code += "MOV ESP, EBP\n";
write.code += "POP EBP\n";
write.code += "RET\n";
write.code += "END_${identifier.name}:\n";
```


Estas linhas de código são responsáveis por desalocar o stack frame da função e retornar para o endereço de memória de onde a função foi chamada.

2.3.1. FuncCallOp

Com as alterações no `FuncDecOp` foi necessário alterar também a Classe `FuncCallOp` que agora não irá mais realizar a execução da função em si, apenas dos argumentos passados para a função.

```
class FuncCallOp extends Node {
    final Identifier identifier;
    final List<Node> arguments;
    FuncCallOp(this.identifier, this.arguments) : super(null);

    @override
    dynamic Evaluate(SymbolTable _table, FuncTable _funcTable) {
        Write write = Write();

        final func = _funcTable.get(identifier.name);

        if (func.parameters.length != arguments.length) {
            throw Exception(
                'Function ${identifier.name} expects ${func.parameters.length} arguments,
but ${arguments.length} were given');
        }

        for (var i = arguments.length - 1; i >= 0; i--) {
            arguments[i].Evaluate(_table, _funcTable);
            write.code += "PUSH EAX\n";
        }

        write.code += "CALL ${identifier.name}\n";
        write.code += "ADD ESP, ${arguments.length * 4}\n";
    }
}
```

Neste código temos a adição de:

```
final func = _funcTable.get(identifier.name);

if (func.parameters.length != arguments.length) {
    throw Exception(
        'Function ${identifier.name} expects ${func.parameters.length} arguments,
but ${arguments.length} were given');
}
```

Estas linhas de código são responsáveis por verificar se a quantidade de argumentos passados para a função é igual a quantidade de parâmetros da função, garantindo que a função será executada corretamente.

```
for (var i = arguments.length - 1; i >= 0; i--) {
    arguments[i].Evaluate(_table, _funcTable);
}
```

```
    write.code += "PUSH EAX\n";  
}
```

Estas linhas de código são responsáveis por passar os argumentos para a função, empilhando os valores na stack de memória.

```
write.code += "CALL ${identifier.name}\n";  
write.code += "ADD ESP, ${arguments.length * 4}\n";
```

Estas linhas de código são responsáveis por chamar a função e desalocar os argumentos da stack de memória.

2.3.2. ReturnOp

A Classe `ReturnOp` foi criada para possibilitar o retorno de valores de funções.

```
class ReturnOp extends Node {  
    final Node expr;  
    ReturnOp(this.expr) : super(null);  
  
    @override  
    dynamic Evaluate(SymbolTable _table, FuncTable _funcTable) {  
        Write write = Write();  
        expr.Evaluate(_table, _funcTable);  
        write.code += "MOV ESP, EBP\n";  
        write.code += "POP EBP\n";  
        write.code += "RET\n";  
    }  
}
```

Neste código temos a adição de:

```
expr.Evaluate(_table, _funcTable);
```

Esta linha de código é responsável por avaliar a expressão que será retornada pela função.

```
write.code += "MOV ESP, EBP\n";  
write.code += "POP EBP\n";  
write.code += "RET\n";
```

Estas linhas de código são responsáveis por desalocar o stack frame da função e retornar para o endereço de memória de onde a função foi chamada.

3. Teste de Execução

Para testar o compilador foi utilizado o seguinte código em Lua:

```
function fatorial(n)  
    if n == 0 then  
        return 1  
    else
```

```

        return n * fatorial(n - 1)
    end
end

print(fatorial(5))

```

resultando no seguinte código asm:

```

; constantes
SYS_EXIT equ 1
SYS_READ equ 3
SYS_WRITE equ 4
STDIN equ 0
STDOUT equ 1
True equ 1
False equ 0

segment .data

formatin: db "%d", 0
formatout: db "%d", 10, 0 ; newline, nul terminator
scanint: times 4 db 0 ; 32-bits integer = 4 bytes

segment .bss ; variaveis
    res RESB 1

section .text
    global main
    extern scanf
    extern printf
    extern fflush
    extern stdout

; subrotinas if/while

binop_je:
    JE binop_true
    JMP binop_false

binop_jg:
    JG binop_true
    JMP binop_false

binop_jl:
    JL binop_true
    JMP binop_false

binop_false:
    MOV EAX, False
    JMP binop_exit

binop_true:

```

```

    MOV EAX, True
binop_exit:
    RET

main:

    PUSH EBP ; guarda o base pointer
    MOV EBP, ESP ; estabelece um novo base pointer

; codigo gerado pelo compilador abaixo
JMP END_fatorial
fatorial:
PUSH EBP
MOV EBP, ESP
IF_22:
MOV EAX, 0
PUSH EAX
MOV EAX, [EBP + 8]
POP EBX
CMP EAX, EBX
CALL binop_je
CMP EAX, False
JE ELSE_22
MOV EAX, 1
MOV ESP, EBP
POP EBP
RET
JMP EXIT_22
ELSE_22:
MOV EAX, 1
PUSH EAX
MOV EAX, [EBP + 8]
POP EBX
SUB EAX, EBX
PUSH EAX
CALL fatorial
ADD ESP, 4
PUSH EAX
MOV EAX, [EBP + 8]
POP EBX
IMUL EAX, EBX
MOV ESP, EBP
POP EBP
RET
EXIT_22:
MOV ESP, EBP
POP EBP
RET
END_fatorial:
MOV EAX, 5
PUSH EAX
CALL fatorial

```

```
ADD ESP, 4
PUSH EAX
PUSH formatout
CALL printf
ADD ESP, 8
; interrupcao de saida (default)

    PUSH DWORD [stdout]
    CALL fflush
    ADD ESP, 4

    MOV ESP, EBP
    POP EBP

    MOV EAX, 1
    XOR EBX, EBX
    INT 0x80
```

Que ao ser compilado com os seguintes comandos em um sistema linux:

```
nasm -f elf -o program.o program.asm
gcc -m32 -o program program.o -no-pie
./program
```

Resulta na saída 120, que é o fatorial de 5.

```
./program - Parrot Terminal
File Edit View Search Terminal Help
SAM-SYSTEM
SAM-SYSTEM.zip
shadow
step.txt
teste.o
trafego.pcap
[antoniom@parrot]~[/Desktop]
$ sudo rm -rf program.asm
[sudo] password for antoniom:
[antoniom@parrot]~[/Desktop]
$ sudo nano program.asm
[antoniom@parrot]~[/Desktop]
$ nasm -f elf -o program.o program.asm
[antoniom@parrot]~[/Desktop]
$ gcc -m32 -o program program.o -no-pie
/usr/bin/ld: warning: program.o: missing .note.GNU-stack section implies executable stack
/usr/bin/ld: NOTE: This behaviour is deprecated and will be removed in a future version of the linker
[antoniom@parrot]~[/Desktop]
$ ./program
120
```

4. Conclusão

A versão 3.1 do compilador desenvolvido em Dart para a linguagem Lua introduziu várias melhorias e novas funcionalidades, ampliando significativamente suas capacidades. As principais mudanças incluem a adição de suporte para funções, a implementação de geração de código em .asm e a reorganização da SymbolTable para melhor gerenciamento de variáveis locais e parâmetros de função.

4.1 Destaques das Melhorias:

1. Tokenização Avançada:

- Adição de novos tokens (function, RETURN, comma) para suportar a definição e chamada de funções.
- Permite a construção de programas mais complexos e estruturados, incluindo funções recursivas.

2. Gerenciamento de Funções:

- Introdução da FuncTable para armazenamento e recuperação de definições de funções.
- Atualizações na SymbolTable para alocação de variáveis locais no stack de memória, utilizando a função setLocalFunction.

3. Geração de Código em .asm:

- Transformação dos operadores para gerar código assembly correspondente, delegando a execução das operações para o código gerado.
- Implementação de um mecanismo para criar e gerenciar stack frames de funções, garantindo o correto armazenamento e recuperação de variáveis locais e parâmetros.

4. Parsing e Execução de Funções:

- O parser agora suporta parsing de funções e retornos, permitindo a definição de funções complexas.
- As classes FuncDecOp, FuncCallOp e ReturnOp foram adicionadas para suportar a definição, chamada e retorno de funções, respectivamente.

Estas mudanças tornam o compilador mais robusto e eficiente, permitindo a compilação e execução de programas Lua mais complexos em um ambiente de baixo nível. O suporte para funções e a geração de código assembly são passos significativos para um compilador que pode ser usado em ambientes de produção, oferecendo maior flexibilidade e desempenho.