Implementação de funções no compilador

Lógica da Computação, Insper

2024

Aluno: Victor Luís Gama de Assis
Professor: Raul Ikeda Gomes da Silva

Introdução

Ao longo do primeiro semestre de 2024 foi desenvolvido um compilador de um subset da linguagem de programação Lua em Python. Esse compilador é constituído por 6 etapas: preprocessamento, tokenização, análise sintática, análise semântica, geração de código e geração do executável. O objetivo desse relatório é apresentar o que foi feito para viabilizar funções capazes de fazer recursão na etapa de geração de código assembly. As mudanças foram realizadas principalmente nos nodes da AST - estrutura de dado que resulta do analisador sintático - e na symbol table.

Implementação

A implementação do gerador de código constituiu em remoção do código do interpretador, refatoração dos nodes da AST e refatoração da Symbol Table.

Remoção do código do interpretador

Ao longo do desenvolvimento das etapas de preprocessamento, tokenização e análise sintática foi desenvolvido um interpretador que executava em profundidade a AST. Esse interpretador possibilitava que as etapas anteriores fossem testadas em um ambiente em produção - a alternativa seria ir direto para um gerador de assembly - além de servir para fins de aprendizado, já que fazia parte do escopo da matéria entender o funcionamento de um interpretador. Por outro lado, as features que realizavam a interpretação do código *.lua* não são mais necessárias uma vez que o objetivo final passou a ser gerar um executável.

Em todos os nodes foi apagado o retorno dos métodos *.evaluate()*. Uma vez que um executável utiliza diretamente endereços da memória e a ULA para acessar e processar informação, não há necessidade de acessar informação retornada dos nodes. Consequentemente, também foi apagado as atribuições dos dados retornados das chamadas de evaluate a variáveis. Também foram apagadas a maioria das chamadas de raise error dentro dos nodes, já que elas já seriam alertadas em tempo de execução pelo próprio computador quando o processo tentasse realizar alguma operação inválida. A única que permaneceu foi a que avaliava se a quantidade de argumentos passados para uma chamada de função correspondia a quantidade esperada em sua definição.

Por fim, todo o código existente na FuncDecNode e FuncCallNode foi apagado. Como será explicado na próxima sessão, o comportamento de ambos os nodes alterou completamente, de modo que foi mais fácil simplesmente reimplementá-los do zero. Isso implicou em alterações no ReturnNode, BlockNode, AssigmentNode, IdentifierNode, VarDecNode e na implementação e uso da SymbolTable, que serão explicadas a seguir. É importante destacar que essa remoção de código do interpretador já deveria ter sido feita no momento que foi implementada a geração de código assembly das outras sintáxes além de funções, porém não o foi por simples esquecimento.

Refatoração dos nodes da AST e da SymbolTable

No interpretador, o FuncDecNode apenas adicionava o nome e a instancia do node da função no FuncTable enquanto FuncCallNode era responsável por iterar nos nodes do código da função os executando. Na medida em que os nodes não mais executam o código - e só faria sentido executar o código de uma função quando ela fosse chamada - e sim traduzem a AST para código ASM, essa estratégia não faz mais sentido. Após refatorar ambos os nodes, o FuncDecNode passa a iterar pelos seus nodes filhos, escrevendo o ASM correspondente, e adiciona um jump em seu início para o seu fim, garantindo que o ASM só será executado quando for chamado. Já o FuncCallNode passa a fazer o CALL para a flag que inicia a função declarada no FuncDecNode.

Além de inverter a lógica de iteração nos filhos do FuncDecNode também trocamos a lógica dos argumentos da função e salvamos e atualizamos o valor do stack pointer. Os argumentos da função, ao invés de serem escritos na SymbolTable local - a criada especificamente para o escopo da função -, passam a ser salvos na pilha e apenas o endereço relativo ao base pointer do escopo da função é salvo na SymbolTable. O método create da SymbolTable ganhou dois parâmetros novos: shift e sign. Ambos são usados apenas no caso de definição de parâmetros de uma função para referenciar o endereço relativo ao base pointer local em que o FuncCallNode escreveu o valor dos parâmetros. Como venho dizendo, também salvamos o valor do base pointer global na pilha e atualizamos o registrador EBP com o endereço do stack pointer. Isso garante que a função tenha um escopo próprio, com base e stack pointer próprios.

```python
class SymbolTable:

    def __init__(self):
        self.table = {}
        self.address = 4

    def create(self, key, shift=0, signal=1):
        if key in self.table:
            raise RuntimeError(f'Key {key} already created.')
        self.table[key] = (self.address + shift) * signal
        self.address += 4

    def get(self, key):
        return self.table[key]

    def set(self, key, address):
        if key in self.table:
            self.table[key] = address
        else:
            raise RuntimeError(f'Key {key} does not exist.')
```

O ReturnNode da evaluate em seu filho e cria o código ASM que retorna o valor no EBP para o ESP, responsável pelo endereço do stack pointer. Consequentemente, também é necessário retirar o valor do base pointer da pilha e salvar no EBP. Por fim, escreve o comando RET, voltando a execução do código para a instrução logo após o CALL. Em resumo, o ReturnNode encerra o escopo da função e retorna o escopo global. Como o BlockNode não interpreta mais o ReturnNode - ou qualquer de seus filhos - não é mais necessário o condicional que checava se o node filho era do tipo return, pode-se simplesmente apagá-lo.

Por fim, uma única alteração é realizada tanto no AssigmentNode, como no IdentifierNode e no VarDecNode. Como agora os argumentos da função estão em um endereço relativos ao base pointer local, é necessário checar se o identifier é de função ou global. Caso seja de função, o sinal do endereço relativo deve ser "+" ao invés de "-" já que os argumentos são adicionados na pilha antes do EBP ser atualizado. Um exemplo de como isso pode ser implementado é "***asm_code = f'MOV [EBP-{abs(address)}], EAX\n' if address > 0 else f'MOV [EBP+{abs(address)}], EAX***", sendo que address é maior que zero quando o escopo é global e menor que zero quando o escopo é local.

Teste

Para realizar o teste foi desenvolvido um código em lua que realiza o cálculo de fatorial recursivamente. Além disso, sua chamada é realizada duas vezes: uma usando uma variável global como parâmetro e outra utilizando um valor hardcoded. Por fim, também foi criada uma variável global com o mesmo nome do argumento da função no escopo local para checar se o compilador consegue diferenciar ambos os contextos.

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

local a
a = 5

local n = 100000

print(factorial(a)) -- 120
print(factorial(0)) -- 1
print(n)
```

O código foi compilado e executado utilizando um arquivo compile.sh, através do comando ./compile.sh test.lua.

```bash
#!/bin/bash
# Extract the filename without the extension
filename=$(basename -- "$1")
extension="${filename##*.}"
filename="${filename%.*}"
# Compile the Lua file
python3 main.py "$1"
# Assemble with NASM
nasm -f elf -o "$filename.o" "$filename.asm"
# Link with GCC
gcc -m32 -no-pie -o "$filename" "$filename.o"
# Run the executable
./"$filename"
# Remove temporary files
rm "$filename.o" "$filename" "$filename.asm"
```

Resultados:

```
victor@linux:~/insper/logica/lua-compiler$ ./compile.sh test.lua
120
1
100000
```