Insper

Lógica da Computação

Aula 20

Raul Ikeda

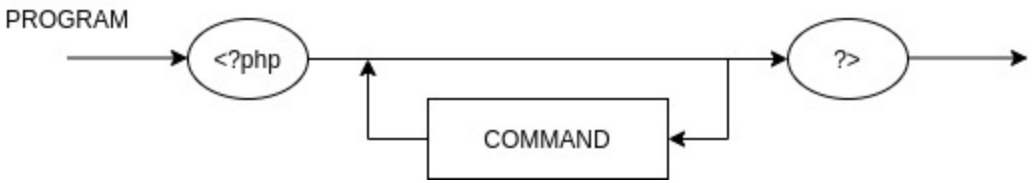
1º semestre de 2020

Esta Aula

- Geração de Código

Pequena Correção

Vamos realizar uma pequena correção:



Compilador (Aproximado):

```
def parseProgram():
    result = Commands(value='',children=[])
    if token == '<?php':
      selectNext()
      while token != '?>':
        Commands.children.append(Parser.parseCommand())
      selectNext()
    else
        raise Error
    return result
```

Corrigir e *taggear* como 2.3.X ainda para poder usar no próximo roteiro.

Gerando um Executável

- Para gerar um arquivo executável precisamos de alguns elementos:
  - Plataforma alvo.
  - Código Assembly traduzido par OPCODE da plataforma alvo
  - Cabeçalho dizendo para o SO alvo como executar o arquivo.
- Para ter o código **assembly**, é preciso gerá-lo no compilador na execução da AST.
- As instruções **assembly** dependem da platafora alvo escolhida.
- Para traduzir para OPCODE, podemos usar **assemblers** (MASM, NASM, GAS, etc).
- Para tornar o objeto executável devemos usar um linker (ld, gcc, etc), dependendo do OS alvo.

Modificando a AST

Ao invés de interpretar a AST, agora vamos fazer com que a AST escreva o correspondente às instruções Assembly. Vamos implementar as seguintes funcionalidades:

- operators (int e boolean)
- variables (int e boolean)
- assignment
- print
- while/if

Limitações (não serão implementados):

- string
- readline
- otimização de código

Ideia da implementação em x86

- Usaremos código NASM x86 (32 bits).
- Trocar o código interpretado na AST por instruções assembly em string.
- Criar uma classe que irá escrever as instruções assembly em um arquivo. Criar um método estático para receber a string das instruções.
- Os nós devem "escrever" o código gerado usando o método da classe acima.
- Será fornecido no Blackboard um arquivo base com cabeçalho, rodapé, constantes, rotinas de print e de comparações.
- IMPORTANTE: como sempre, cada nó se preocupa somente com o seu próprio código e NÃO interfere na geração do código dos filhos. Apenas confia no Evaluate().

IntVal

- **TODO** nó que retorna valor o fará em **EBX**. Isso vale para qualquer nó.

```php
<?php
  3; /* IntVal: Código hipotético */
?>
```

```
MOV EBX, 3 ; Evaluate do IntVal
```

BinOp

- BinOp precisa guardar o valor do nó esquerdo na pilha antes de executar o nó direiro (os dois vão retornar em EBX).

```php
<?php
  4 + 3; /* BinOp: Código hipotético */
?>
```

```
; Código gerado pelo IntVal (direita):
MOV EBX, 4 ; Evaluate() do filho IntVal da esquerda

; Código gerado pelo BinOp (depois do return do filho):
PUSH EBX ; O BinOp guarda o resultado na pilha

; Código gerado pelo IntVal (direita):
MOV EBX, 3 ; Evaluate() do filho IntVal da direita

; Código gerado pelo BinOp (depois do return do filho):
POP EAX ; O BinOp recupera o valor da pilha em EAX
ADD EAX, EBX ; O BinOp executa a operação correspondente
MOV EBX, EAX ; O BinOp retorna o valor em EBX (sempre EBX)
```

Identifier

- As variáveis vão ficar na pilha.
- Assumir tamanho fixo de 4 bytes (DWORD).
- Na primeira atribuição é preciso alocar a pilha. Também é preciso guardar na Symboltable quantos bytes estará deslocado do EBP para referências futuras.

```php
<?php
  $x = 3;
  $x = 5;
?>
```

```
; primeira atribuição
PUSH DWORD 0 ; alocação na primeira atribuição
MOV EBX, 3 ; Evaluate() do filho da direita
MOV [EBP-4], EBX; resultado da atribuição - não há return

; segunda atribuição
MOV EBX, 5 ; Evaluate() do filho da direita
MOV [EBP-4], EBX; resultado da atribuição
```

Exemplo: Identifier + BinOp

```php
<?php
  $x = 10;
  $y = $x - 5;
?>
```

```
; Atribuição em x
PUSH DWORD 0 ; alocação na primeira atribuição
MOV EBX, 10 ; Evaluate() do filho da direita
MOV [EBP-4], EBX; resultado da atribuição

; Atribuição em y
PUSH DWORD 0 ; alocação na primeira atribuição
; Código gerado pelo Evaluate() do BinOp à direita do Assigment
MOV EBX, [EBP-4] ; Evaluate() do Identifer à esquerda do BinOp
PUSH EBX ; O BinOp guarda o resultado na pilha
MOV EBX, 5 ; Evaluate() do IntVal à direita do BinOp
POP EAX ; O BinOp recupera o valor da pilha em EAX
SUB EAX, EBX ; O BinOp executa a operação correspondente
MOV EBX, EAX ; O BinOp retorna o valor em EBX (sempre EBX)

MOV [EBP-8], EBX; resultado da atribuição
```

Print

- Para realizar o print, basta chamar a subrotina *print*. Note que o valor já vai estar em EBX.

```php
<?php
  $x = 10;
  echo $x;
?>
```

```
; Atribuição em x
PUSH DWORD 0 ; alocação na primeira atribuição
MOV EBX, 10 ; Evaluate() do filho da direita
MOV [EBP-4], EBX; resultado da atribuição

MOV EBX, [EBP-4] ; Evaluate do Identifier, único filho do print

PUSH EBX ; Empilhe os argumentos
CALL print ; Chamada da função
POP EBX ; Desempilhe os argumentos
```

while/if

- É preciso utilizar *label* e *jump* para a implementação em assembly.
- Como podem existir várias instruções repetidas, não podemos utilizar o mesmo label.
- Portanto é preciso utilizar um identificar universal (número) que será atribuído na criação de um nó. Esse idenficador fará parte do label para evitar a repetição dos nomes.

```
LOOP_34: ; o unique identifier do nó while é 34

; instruções do filho esquerdo do while - retorna o resultado em EBX

CMP EBX, False ; verifica se o teste deu falso
JE EXIT_34 ; e sai caso for igual a falso.

; instruções do filho direito do while.

JMP LOOP_34 ; volta para testar de novo
EXIT_34:
```

Criando o Executável

- Uma vez criado o arquivo com código assembly, chegou a vez de montar usando um assembler e colocar o cabeçalho de execução usando um linker.
- Para montar x86 + Linux:

```
$ nasm -f elf32 -F dwarf -g program.asm
$ ld -m elf_i386 -o program program.o
```

- Mais detalhes em: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
- Para debugar o código final: **gdb**

Atividade: Roteiro 8

- Roteiro Impresso ou PDF no Blackboard.

Próxima Aula

- P vs NP
- NP-Completude
- Complexidade de Espaço

Referências:

- Sipser Cap. 7 e 8
- Hopcroft Cap. 10 e 11.2