Compiladores, 2023/2024

Trabalho prático, parte 4
– Funções –

Fernando Lobo

1 Introdução

Nesta parte do trabalho prático, vamos extender a linguagem Sol para permitir a declaração e uso de funções, bem como a declaração e uso de variáveis locais.

Como consequência, a tabela de símbolos terá de suportar enquadramentos encadeados (nested scopes) como foi explicado nas aulas teóricas e ser capaz de guardar informação relevante para todo o tipo de símbolos (variáveis globais, variáveis locais, nomes de funções, argumentos de funções).

Devido à introdução de funções, a linguagem Sol terá de permitir mais dois tipos de instrução, nomeadamente:

- chamada de função
- return

Por outro lado, teremos de acrescentar novas instruções à máquina virtual de modo a poder suportar a execução de funções.

A chamada de uma função poderá aparecer no programa fonte antes da declaração da função, sem que isso cause qualquer erro.

2 Alterações à linguagem

2.1 Programa

Um programa em Sol passa agora a ser constituído por uma sequência de zero ou mais declarações de variáveis globais, seguido de uma sequência de uma ou mais declarações

de funções, sendo que uma dessas funções terá de se chamar `main`, o ponto de entrada para a execução do programa.

2.2 Declaração de função

Uma declaração de função é feita indicando o tipo de retorno (`void`, `int`, `real`, `bool` ou `string`), seguido do nome da função, seguido de um parêntesis a abrir, seguido de uma lista de definição de argumentos separados por vírgula, seguido de um parêntesis a fechar, seguido de um bloco.

O tipo `void` indica a ausência de valor de retorno, tal como em C e Java.

A definição de argumento é especificada indicando o tipo e o seu nome. O nome da função e argumento(s) obedecem às mesmas regras lexicais usadas para as declarações de variáveis. Eis um exemplo com a declaração de 2 funções:

```
void hello( string s )
begin
    print "Hello " + s;
end

int max( int a, int b )
begin
    if a > b then return a;
    else return b;
end
```

O compilador deverá fazer a verificação de tipos. Quando uma função é definida como retornando um valor de determinado tipo, o compilador deve garantir que de facto a função retorna um valor, e que esse valor tem um tipo de dados consistente com o tipo de dados declarado.

Consistência de tipos significa que os tipos têm de ser iguais, havendo apenas a excepção de algo declarado como sendo do tipo `real` poder receber um valor do tipo `int`, devendo neste caso o compilador emitir código para fazer a conversão usando a instrução `itod` da máquina virtual.

No caso da função ser void, a existência de um return explícito por parte do programador é opcional.

2.3 Novas instruções

2.3.1 Chamada de função

Uma chamada de função seguido de um ponto e vírgula, passa a ser uma instrução válida da linguagem Sol, se essa função estiver declarada como retornando void. Se o valor de retorno não for void deixa de ser uma instrução válida. Por exemplo, mediante as declarações de funções mencionadas anteriormente, apenas a 1ª linha do seguinte excerto de código corresponde a uma instrução válida:

```
hello("Maria");     // é válido porque hello retorna void
max(a,b);           // não é válido porque max retorna int
```

Note porém, que `max(a,b)` é considerado uma expressão válida (ver Secção 2.4 adiante).

Return

Esta instrução permite terminar a execução de uma função, retornando um valor que deverá ser do mesmo tipo que foi especificado na definição da função. A sintaxe da instrução é a palavra `return` seguida de uma expressão, seguida de um ponto-e-vírgula. A expressão é opcional, porque podemos querer terminar a execução de uma função sem retornar qualquer valor (função void). Este comportamento é análogo ao que acontece em C e Java. Exemplos:

```
return a+b;
return ;
```

Bloco (alteração)

A sintaxe de um bloco passa a ser: palavra reservada `begin`, seguido de zero ou mais declarações de variáveis, seguido de zero ou mais instruções, seguido de palavra reservada `end`. Exemplo:

```
begin
    bool b1 = true, b2 = false;
    int x;
    x = 5;
    print b1 and b2;
end
```

2.4 Novo tipo de expressão

Chamada de função

Uma chamada de função passa a ser uma expressão válida. Uma chamada de função é especificada tal como em C e Java: nome da função, seguido de um parêntesis a abrir, seguido de uma lista (porventura vazia) de argumentos separados por vírgula, seguido de um parêntesis a fechar. Cada argumento, por sua vez, é também uma expressão. Exemplo de expressão válida:

```
max(a,max(b,c))
```

Durante a verificação de tipos, deve ser feita também a verificação de consistência entre o tipo de dados dos argumentos/parâmetros formais e dos argumentos/parâmetros actuais, bem como do seu número. No exemplo dado, a função max foi declarada como tendo 2 argumentos do tipo int. Logo, quando a função é chamada terá de ser garantido que lhe estamos a passar duas (e apenas duas) expressões, e cada uma delas terá de ser de um tipo consistente com a definição.

A passagem de parâmetros é feita por valor. Isto é, em tempo de execução, uma cópia do valor do argumento é copiado para o parâmetro formal da função.

3 Novas instruções da máquina virtual

De modo a suportar a execução de funções, a máquina virtual vai passar a ter um registo especial chamado FP (Frame Pointer) que aponta para a base do *frame* em execução. Para além disso devem ser acrescentadas 7 novas instruções à máquina virtual, todas elas contendo um argumento inteiro. O nome e descrição destas novas instruções é apresentado na tabela abaixo.

OpCode	**Argumento**	**Descrição**
lalloc	inteiro n	*local memory allocation*: aloca n posições no topo do stack para armazenar variáveis locais. Essas n posições de memória ficam inicializadas com o valor NIL.
lload	inteiro *addr*	*local load*: empilha o conteúdo de $Stack[FP + addr]$ no stack.
lstore	inteiro *addr*	*local store*: faz pop do stack e guarda o valor em $Stack[FP + addr]$
pop	inteiro n	desempilha n elementos do stack

call	inteiro *addr*	Cria um novo frame no stack, que passará a ser o frame currente. Para tal terá de guardar o estado da máquina virtual (isto é, empilha o valor do registo FP, actualiza o valor de FP para apontar para a base do novo frame, e empilha o endereço de retorno: o endereço da instrução imediatamente após o call). Depois actualiza o *instruction pointer* IP de modo a que a próxima instrução a ser executada seja aquela que se encontra na posição *addr* do array de instruções.
retval	inteiro n	*return from non-void function*: faz $x = pop()$, desempilha o espaço reservado para as variáveis locais usadas pela função, restaura o estado da máquina virtual, desempilha os n argumentos do stack, e depois empilha x
ret	inteiro n	*return from void function*: desempilha o espaço reservado para as variáveis locais usadas pela função, restaura o estado da máquina virtual, e desempilha os n argumentos do stack

NOTA: O modo como a máquina virtual suporta a execução de funções será explicado detalhadamente nas aulas teóricas de 7 e 9 de Maio, acompanhado de exemplos de execução da VM, passo a passo. É fundamental perceberem muito bem esse mecanismo.

4 Alterações ao tAssembler

Para além de acrescentarem estas 7 novas instruções à máquina virtual, deverão também modificar o vosso tAssembler realizado no primeiro trabalho de modo a que possa suportar estas novas instruções. Note que para o assembler, a instrução `call` tem um argumento que deve ser um label (tal como num jump), e as instruções `lload` e `lstore` podem ter como argumento um inteiro negativo.

5 Extra: Para quem quer ter 19 ou 20 valores à disciplina

Quem quiser ter nota igual ou superior a 19 valores à disciplina, terá ainda de implementar o seguinte:

- Suportar a passagem de parâmetros por referência em funções, algo que permite que o conteúdo de uma variável que é passada como argumento, possa ser alterado pela função.
- Suportar arrays com várias dimensões.

A implementação destes aspectos não chegou a ser visto nas aulas, mas trata-se de algo que não é muito complicado de se fazer dado o conhecimento que já adquiriram, e que pode ser complementado com a consulta a bibliografia recomendada para a disciplina. Bem sei que o tempo é apertado, pelo que estas funcionalidades extra são apenas exigidas para quem quiser ter uma classificação final superior a 18.

Notem que para a realização desta parte têm liberdade para definir regras para a sintaxe e semântica destas novas funcionalidades, e têm também liberdade para acrescentar novas instruções à máquina virtual caso achem necessário.

6 Condições de realização

O trabalho deve ser realizado em grupo, de acordo com as inscrições em grupo do laboratório. Deverão submeter o vosso código num ficheiro ZIP por via eletrónica, através da tutoria, até às 23:59 do dia 19/Mai/2024. (**NOTA:** Não serão aceites entregas fora de prazo, nem que seja por um só minuto. Não devem deixar a submissão para os últimos instantes. Podem submeter o vosso trabalho várias vezes, e só a última submissão é que conta.)

O código ZIP deverá conter a gramática em ANTLR, bem como todo o código Java desenvolvido.

- O trabalho deve ser feito em Java usando o ANTLR 4.
- O ficheiro ZIP deverá conter uma pasta chamada `src` onde está todo o código que desenvolveram.
- Também deverá conter uma pasta chamada `inputs` onde deverão estar vários exemplos de ficheiros `.sol` que usaram para testar o vosso código
- O vosso compilador deverá chamar-se `solCompiler`. Deverá receber como input um ficheiro com extensão `.sol` e gerar como output 2 ficheiros: um ficheiro `.tbc` com bytecodes pronto a ser executado pela vossa tVM, e um ficheiro `.tasm` que deverá poder ser dado como input ao tAssembler.
- Os ficheiros de output devem ser criados na mesma pasta onde está o ficheiro de input, e devem ter exactamente o mesmo nome com excepção da extensão. Por exemplo, se o input for `Teste/in1.sol`, os ficheiros de output gerados deverão ser `Teste/in1.tbc` e `Teste/in1.tasm`

- O vosso assembler alterado para poder suportar as novas instruções da VM. O assembler deverá chamar-se `tAssembler` e deverá receber como input um ficheiro `.tasm` e produzir como output um ficheiro `.tbc` com o mesmo nome
- A vossa VM alterada que suporte a execução das novas instruções descritas neste documento. A VM deverá chamar-se `tVM` e deverá receber como input um ficheiro `.tbc`
- Podem (e devem) correr os bytecodes gerados pela implementação da vossa máquina virtual, certificando-se que o programa Sol faz aquilo que é suposto fazer.
- A gramática em ANTLR para a linguagem Sol deve ter o nome `Sol.g4`
- A gramática em ANTLR para a linguagem Tasm deve ter o nome `Tasm.g4`

Apenas é necessário que 1 dos elementos do grupo submeta o trabalho na tutoria.

7 Validação e avaliação

Os trabalhos serão avaliados de acordo com a clareza e qualidade do código implementado, pela correcta implementação das diversas funcionalidades, e pelo desempenho individual durante a validação.

A ponderação deste trabalho na nota final é de 25%.

A parte extra referida na Secção 5 é considerada à parte, e é usada apenas para a obtenção da nota final da disciplina. Ou seja, um aluno que tenha nota máxima em todos os trabalhos mas que não implemente o que está na Secção 5 ficará com 18 valores de nota final.

Apêndice A Exemplos

A.1 Programa com erros semânticos

```
1  void mainnn()
2  begin
3      int n = fun(1,2,3);
4      fun(1,2);
5      hello("Maria");
6      hello(5);
7  end
8
9  void hello( string s )
10 begin
11    print "Hello " + s;
12 end
13
14 real zzz( int x )
15 begin
16    return x + 1;
17 end
18
19 bool xpto()
20 begin
21    print "ola";
22 end
23
24 int fun( int x, int y )
25 begin
26    bool b = xpto;
27    if x < 1 then return x + y;
28 end
```

Output que obtive pelo meu solCompiler:

```
line 29:0 error: missing main() function
line 3:12 error: 'fun' has 2 arguments, not 3
line 4:4 error:  value of 'fun' should be assigned to a variable
line 6:10 error: '5' is of type int, but type string expected
line 22:0 error: missing return in function 'xpto'
line 26:12 error: 'xpto' is not a variable
line 28:0 error: missing return in function 'fun'
inputs-t4/t4erros.sol has 7 semantic errors
```

A.2 t4locals.sol

```
1  void main()
2  begin
3     int a = 1; int b;
4     begin
5        int c = 2;
6        begin
7           int d = 3; int e;
8           begin
9              int f = 4;
10          end
11          e = 5;
12       end
13    end
14    b = 6;
15    begin
16       int g = 7;
17    end
18 end
```

Ficheiro Tasm gerado pelo solCompiler:

```
L0: call L2
L1: halt
L2: lalloc 1
L3: iconst 1
L4: lstore 2
L5: lalloc 1
L6: lalloc 1
L7: iconst 2
L8: lstore 4
L9: lalloc 1
L10: iconst 3
L11: lstore 5
L12: lalloc 1
L13: lalloc 1
L14: iconst 4
L15: lstore 7
L16: pop 1
L17: iconst 5
L18: lstore 6
```

```
L19: pop 2
L20: pop 1
L21: iconst 6
L22: lstore 3
L23: lalloc 1
L24: iconst 7
L25: lstore 4
L26: pop 1
L27: pop 2
L28: ret 0
```

A.3 t4-exAula.sol

```
1  int sqr( int x )
2  begin
3      return x * x;
4  end
5
6  int sqrsum( int a, int b )
7  begin
8      int s;
9      s = sqr(a + b);
10     return s;
11 end
12
13 void main()
14 begin
15     print sqrsum(3,2);
16 end
```

Ficheiro Tasm gerado pelo solCompiler:

```
L0: call L14
L1: halt
L2: lload -1
L3: lload -1
L4: imult
L5: retval 1
L6: lalloc 1
L7: lload -2
L8: lload -1
L9: iadd
L10: call L2
L11: lstore 2
L12: lload 2
L13: retval 2
L14: iconst 3
L15: iconst 2
L16: call L6
L17: iprint
L18: ret 0
```

Execução dos bytecodes pela tVM dá:

25

A.4 t4max.sol

```
1  int max( int a, int b )
2  begin
3     if a > b then return a;
4     else return b;
5  end
6
7  void main()
8  begin
9     int x = 3, y = 5;
10    int z = max(x,y);
11    print z;
12 end
```

Ficheiro Tasm gerado pelo solCompiler:

```
L0: call L11
L1: halt
L2: lload -1
L3: lload -2
L4: ilt
L5: jumpf L9
L6: lload -2
L7: retval 2
L8: jump L11
L9: lload -1
L10: retval 2
L11: lalloc 2
L12: iconst 3
L13: lstore 2
L14: iconst 5
L15: lstore 3
L16: lalloc 1
L17: lload 2
L18: lload 3
L19: call L2
L20: lstore 4
L21: lload 4
L22: iprint
L23: pop 3
L24: ret 0
```

Execução dos bytecodes pela tVM dá:

5

A.5 t4factorial.sol

```
1  void main()
2  begin
3     print fact(3);
4  end
5
6  int fact( int n )
7  begin
8     if n == 0 then return 1;
9     return n * fact(n-1);
10 end
```

Ficheiro Tasm gerado pelo solCompiler:

```
L0: call L2
L1: halt
L2: iconst 3
L3: call L6
L4: iprint
L5: ret 0
L6: lload -1
L7: iconst 0
L8: ieq
L9: jumpf L12
L10: iconst 1
L11: retval 1
L12: lload -1
L13: lload -1
L14: iconst 1
L15: isub
L16: call L6
L17: imult
L18: retval 1
```

Execução dos bytecodes pela tVM dá:

```
6
```

Outputs obitos pela execução da VM em modo *trace* serão colocados à vossa disposição na tutoria.