Compiladores, 2023/2024

Trabalho prático, parte 3
– Variáveis globais, instruções de controlo de fluxo –

1 Introdução

Neste trabalho vamos extender a linguagem Sol para permitir declarações de variáveis globais, instrução de afectação, e instruções de controlo de fluxo. Especificamente, para além da instrução print existente no trabalho 2, passaremos a ter as seguintes instruções:

- afectação
- bloco
- while
- for
- if
- vazia
- break

2 Alterações à linguagem

2.1 Programa

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

2.2 Declaração de variáveis

A declaração de variáveis é feita de modo análogo ao que é feito em C: indica-se um dos 4 tipo de dados suportados pela linguagem Sol (`bool`, `int`, `real`, `string`), que passam a ser palavras reservadas [1], seguido de uma sequência de nomes separados por vírgula (em que opcionalmente se pode inicializar a variável com um valor usando o sinal `=`), seguido de um ponto-e-vírgula.

O nome da variável obedece à mesma regra que é usada na linguagem C. Isto é, deve começar por uma letra ou por um underscore, e depois pode vir uma sequência de letras, digítos, ou underscores. Exemplo:

```
int i = 0, n;
bool b = true;
```

Na fase de análise semântica deve ser feito a verificação de tipos. Para além disso, deve ser reportado um erro se fizermos referência a uma variável não declarada, ou se tentarmos declarar uma varável que já foi declarada anteriormente.

Uma variável que não é inicializada fica na máquina virtual com o valor `NIL`, e o acesso a uma variável nessas condições deve gerar um erro de runtime pela máquina virtual tVM.

2.3 Novo tipo de expressão

Uma vez que é permitido variáveis, também é obviamente permitido ter variáveis no contexto de expressões. Por exemplo, `n + 1` passa a ser uma expressão sintáticamente válida na linguagem Sol. Na fase de verificação de tipos deve usar as regras especificadas no trabalho anterior, tendo presente que o tipo de uma variável é obtido através da sua declaração.

2.4 Novos tipos de instrução

Afectação

A instrução de afectação permite atribuir um valor a uma variável. A sintaxe é análoga à da linguagem C ou Java: nome da variável, seguido de um sinal de `=`, seguido de uma expressão, seguido de um ponto-e-vírgula. Exemplo: `x = 4;`

[1] Palavras reservadas não podem ser usadas como nomes de variáveis, funções, etc.

Bloco

Um bloco, usualmente também designado por instrução composta, é uma instrução que serve para agrupar instruções. Ao invés das chavetas usadas em C e Java, o início do bloco é especificado com a palavra reservada `begin` e o fim do bloco é especificado com a palavra reservada `end`. O contéudo do bloco é uma sequência de zero ou mais instruções. Exemplo:

```
begin
    print x;
    x = x+1;
end
```

While

A instrução while serve para fazer ciclos, tal como em C e Java. A instrução deve começar com a palavra reservada `while`, seguido de uma expressão, seguido da palavra reservado `do`, seguido de uma instrução. Exemplo:

```
while x < 10 do
begin
    print x;
    x = x+1;
end
```

Na fase de análise semântica deve ser garantido que a expressão de controlo do ciclo while tem de ser do tipo bool.

For

A instrução for também serve para fazer ciclos. A instrução começa com a palavra reservada `for`, seguido de um identificador, seguido do sinal `=`, seguido de uma expressão, seguido da palavra reservado `to`, seguido de outra expressão, seguido da palavra reservada `do`, seguido de uma instrução. Exemplo:

```
for i=1 to n do
    print i;
```

Na fase de análise semântica deve ser garantido que o identificador é uma variável do tipo int, e que as expressões de controlo do ciclo têm de ser do tipo int.

A variável de controlo do ciclo é incrementada de uma unidade a cada iteração do ciclo. O ciclo termina quando o valor da variável de controlo excede o valor da

segunda expressão. O exemplo acima podia ser rescrito com um while da seguinte forma:

```
i=1;
while i <= n do
begin
    print i;
    i = i+1;
end
```

If

A instrução começa com a palavra reservada `if`, seguido de uma expressão, seguido da palavra reservada `then`, seguido de uma instrução, e opcionalmente poderá ter a parte else, em que aparece a palavra reservada `else` seguido de uma instrução. Exemplo:

```
if x < 10 then print "ola";
else begin
         print x;
         x = x+1;
     end
```

Na fase de análise semântica deve ser garantido que a expressão do if tem de ser do tipo bool.

Vazia

Um ponto-e-virgula é uma instrução. (É uma instrução vazia que não faz nada.)

Break

A palavra reservada `break` seguido de um ponto-e-vírgula é uma instrução. A instrução break só pode ocorrer dentro de um ciclo (while ou for), e o seu signifcado é equivalente ao que existe em C e Java: quando há um break, a execução do código salta para a primeira instrução que aparece imediatamente após o fim do ciclo respectivo.

3 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 06/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 estara 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 deveria poder ser dado como input ao tAssembler feito durante a 1º trabalho.
- 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`
- 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 deve ter o nome `Sol.g4`

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

4 Validação e avaliação

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

A ponderação deste trabalho na nota final é de 20%.

Apêndice A Exemplos

A.1 Programa com erros semânticos

O seguinte programa está sintáticamente correcto mas tem vários erros semânticos.

```
int i = 0, n = 10;
real r = 3.24;
int n;
bool b = 5;
string s;

while s <= n do
begin
    print i * i + z;
    i = i + 1;
end
break;
print "Fim!";
```

Output que obtive pelo meu solCompiler:

```
line 3:4 error: n is already defined
line 4:7 error: operator = is invalid between bool and int
line 7:8 error: operator <= is invalid between string and int
line 7:6 error: while expression must be of type bool
line 9:18 error: z is not defined
line 12:0 error: break statement can only occur inside a loop
inputs-t3/t3erros.sol has 6 semantic errors
```

A.2 Exemplo if

```
int n = 11;

if n % 2 == 0
   then print "par";
   else print "impar";
print "Fim!";
```

Ficheiro Tasm gerado pelo solCompiler:

```
L0: galloc 1
L1: iconst 11
L2: gstore 0
L3: gload 0
L4: iconst 2
L5: imod
L6: iconst 0
L7: ieq
L8: jumpf L12
L9: sconst "par"
L10: sprint
L11: jump L14
L12: sconst "impar"
L13: sprint
L14: sconst "Fim!"
L15: sprint
L16: halt
```

Execução dos bytecodes pela tVM dá:

```
impar
Fim!
```

A.3 Exemplo while

```
int i, n;

n = 7;
i = 1;
while i <= 10 do
begin
    print n + " x " + i + " = " + n*i;
    i = i + 1;
end
```

Ficheiro Tasm gerado pelo solCompiler:

```
L0: galloc 2
L1: iconst 7
L2: gstore 1
L3: iconst 1
L4: gstore 0
L5: gload 0
L6: iconst 10
L7: ileq
L8: jumpf L29
L9: gload 1
L10: itos
L11: sconst " x "
L12: sadd
L13: gload 0
L14: itos
L15: sadd
L16: sconst " = "
L17: sadd
L18: gload 1
L19: gload 0
L20: imult
L21: itos
L22: sadd
L23: sprint
L24: gload 0
L25: iconst 1
L26: iadd
L27: gstore 0
```

```
L28: jump L5
L29: halt
```

Execução dos bytecodes pela tVM dá:

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

A.4 Exemplo for

```
int i;
for i=1 to 5 do
    print i*i;
print "Fim!";
```

Ficheiro Tasm gerado pelo solCompiler:

```
L0: galloc 1
L1: iconst 1
L2: gstore 0
L3: gload 0
L4: iconst 5
L5: ileq
L6: jumpf L16
L7: gload 0
L8: gload 0
L9: imult
L10: iprint
L11: gload 0
L12: iconst 1
L13: iadd
L14: gstore 0
L15: jump L3
L16: sconst "Fim!"
L17: sprint
L18: halt
```

Execução dos bytecodes pela tVM dá:

```
1
4
9
16
25
Fim!
```

A.5 Exemplo break

```
/*
 * Computes all prime numbers upto n
 */

int i, j, n = 30;
bool isPrime;
string result = "";

for i = 1 to n do
begin
    // check if i is prime
    isPrime = true;
    for j = 2 to i/2 do
        if i % j == 0 then
        begin
            isPrime = false;
            break;
        end
    if isPrime
       then result = result + " " + i;
end
print "Prime numbers upto " + n + ":" + result;
```

Ficheiro Tasm gerado pelo solCompiler:

```
L0: galloc 3
L1: iconst 30
L2: gstore 2
L3: galloc 1
L4: galloc 1
L5: sconst ""
L6: gstore 4
L7: iconst 1
L8: gstore 0
L9: gload 0
L10: gload 2
L11: ileq
L12: jumpf L51
L13: tconst
L14: gstore 3
```

```
L15: iconst 2
L16: gstore 1
L17: gload 1
L18: gload 0
L19: iconst 2
L20: idiv
L21: ileq
L22: jumpf L37
L23: gload 0
L24: gload 1
L25: imod
L26: iconst 0
L27: ieq
L28: jumpf L32
L29: fconst
L30: gstore 3
L31: jump L37
L32: gload 1
L33: iconst 1
L34: iadd
L35: gstore 1
L36: jump L17
L37: gload 3
L38: jumpf L46
L39: gload 4
L40: sconst " "
L41: sadd
L42: gload 0
L43: itos
L44: sadd
L45: gstore 4
L46: gload 0
L47: iconst 1
L48: iadd
L49: gstore 0
L50: jump L9
L51: sconst "Prime numbers upto "
L52: gload 2
L53: itos
L54: sadd
L55: sconst ":"
L56: sadd
L57: gload 4
L58: sadd
L59: sprint
```

```
L60: halt
```

Execução dos bytecodes pela tVM dá:

```
Prime numbers upto 30: 1 2 3 5 7 11 13 17 19 23 29
```