Compiladores, 2023/2024

Trabalho prático, parte 2
– Linguagem Sol: expressões, consistência de tipos –

1 Introdução

Neste e nos restantes trabalhos da disciplina iremos desenvolver um compilador para uma linguagem imperativa chamada Sol, inventada por mim. Para começar, iremos nos restringir a fazer a compilação de programas que apenas permitem um tipo de instrução: print.

Nos restantes trabalhos iremos acrescentar outro tipo de instruções e outro tipo de funcionalidades à linguagem.

Caso o input não tenha erros, o vosso compilador deve gerar bytecodes para a máquina virtual T, implementada por vós no trabalho anterior.

2 Sintaxe da linguagem

Um programa em Sol é uma sequência de uma ou mais instruções print, cada qual consistindo na palavra reservada `print`, seguido de uma expressão, seguido de um ponto-e-vírgula.

A expressão pode envolver operadores aritméticos, lógicos, e também de concatenação de strings. A sintaxe de uma expressão pode ser descrita de forma indutiva através das seguintes regras.

1. um literal (um inteiro, um real, uma string, true, false) é por definição uma expressão.
2. um operador unário seguido de uma expressão, é uma expressão.
3. uma expressão, seguida de um operador binário, seguido de outra expressão, é também uma expressão.

4. Um parêntesis a abrir, seguido de uma expressão, seguido de um parêntesis a fechar, é também uma expressão.

A linguagem Sol suporta 4 tipos de dados: bool, int, real, string. O tipo bool suporta os valores `true` e `false`. O tipo int suporta inteiros. O tipo real suporta números reais (equivalente a um double em Java). O tipo string suporta sequência de caracteres delimitados por aspas.

Tal como em C e Java, podemos ter comentários de uma só linha (começam com `//` e vão até ao final da linha) ou multi-linha (começam com `/*` e terminam com `*/`).

Para facilitar-vos a vida, dou-vos as regras lexicais em ANTLR que permitem ter este tipo de comentários.

```
SL_COMMENT :   '//' .*? (EOF|'\n') -> skip;  // single-line comment
ML_COMMENT :   '/*' .*? '*/' -> skip ;       // multi-line comment
```

2.1 Exemplo de um programa em Sol

```
/*
   Exemplo de programa em Sol
*/
print 1 + 2 * 3;
print 1 + 2.0 * 3;
print 7 % (1 + 4);
print true and (5 < 3.14159);
print "pi = " + 3.14159;
print "ma" + "ria" == "maria";
```

2.2 Precedência dos operadores

A tabela seguinte apresenta a precedência dos operadores da linguagem Sol. Quanto mais acima na tabela, maior é a precedência do operador (e menor é o número que aparece na 1ª coluna da tabela). Dizer que um operador tem maior precedência que outro, significa que esse operador deve ser avaliado primeiro do que o operador com precedência mais baixa.

Os operadores que aparecem na mesma linha, têm o mesmo nível de precedência. Nesse caso, a ordem pela qual as operações são feitas é da esquerda para a direita, i.e. todos têm associatividade à esquerda.

Precedência	Descrição	Operadores
1	Parênteses	`()`
2	Unário	`- not`
3	Multiplicação e Divisão	`* / %`
4	Soma e Subtração	`+ -`
5	Relacional	`< > <= >=`
6	Igualdade e Desigualdade	`== !=`
7	E lógico	`and`
8	Ou lógico	`or`

Os operadores listados na tabela acima têm o significado usual e são em tudo semelhantes a operadores que existem no C e Java. Note porém que o símbolo usado para alguns deles é diferente, nomeadamente os operadores `not`, `and`, `or` (que em Java são escritos como `!`, `&&`, `||`, respectivamente).

3 Regras sobre tipos de dados

Tal como acontece na maioria das linguagens de programação, a utilização dos operadores acima descritos obedecem a algumas regras, dependendo do tipo de dados do(s) operando(s). Por exemplo, a seguinte instrução,

```
print true - 6;
```

apesar de ser sintaticamente correcta, não é válida porque o operador binário '`-`' não pode ser aplicado quando um dos operandos é do tipo bool. Como tal, o compilador deverá emitir uma mensagem de erro apropriada neste tipo de situação. De seguida apresenta-se as regras sobre os tipos de dados da linguagem Sol.

3.1 Regras para os literais

- um literal inteiro é uma expressão do tipo int
- um literal real é uma expressão do tipo real
- uma string delimitada por aspas é uma expressão do tipo string
- `true` é uma expressão do tipo bool
- `false` é uma expressão do tipo bool

3.2 Regras para operadores unários

op	*expr*	*op expr*
`-`	int	int
`-`	real	real
`not`	bool	bool

Cada uma das linhas da tabela especifica uma regra. Por exemplo, a 1ª linha da tabela acima diz que o operador unário `-` aplicado a uma expressão do tipo int, dá origem a uma expressão do tipo int. Como se pode constatar, o operador unário `-` só pode ser aplicado para expressões do tipo int ou real. Por sua vez o operador unário `not` só pode ser aplicado para expressões do tipo bool.

3.3 Regras para operadores binários

op	$expr_1$	$expr_2$	$expr_1$ *op* $expr_2$
`+ - * / %`	int	int	int
`+ - * /`	int	real	real
`+ - * /`	real	int	real
`+ - * /`	real	real	real
`+`	bool	string	string
`+`	string	bool	string
`+`	int	string	string
`+`	string	int	string
`+`	real	string	string
`+`	string	real	string
`+`	string	string	string
`and or`	bool	bool	bool
`< > <= >=`	int	int	bool
`< > <= >=`	int	real	bool
`< > <= >=`	real	int	bool
`< > <= >=`	real	real	bool
`== !=`	bool	bool	bool
`== !=`	int	int	bool
`== !=`	int	real	bool
`== !=`	real	int	bool
`== !=`	real	real	bool
`== !=`	string	string	bool

Novamente, cada linha da tabela acima especifica uma regra. A aplicação de operadores fora do contexto acima descrito deverá originar um erro semântico de inconsistência de tipos (não de parsing), que deve ser reportado pelo compilador de forma apropriada.

Como referido anteriormente, o significado dos operadores é em tudo idêntico ao da linguagem Java. Note por exemplo que o operador `+` tanto pode ser usado com o significado de soma, quando os operandos são valores numéricos (int ou real), como também pode ser usado para fazer concatenação de strings, quando pelo menos um dos operandos é uma string, sendo que o compilador deve gerar código para fazer conversões de tipos se necessário. Por exemplo, na expressão:

```
"pi = " + 3.14159
```

a expressão do lado esquerdo é uma string e a expressão do lado direito é um real. Como tal, é suposto o compilador gerar código para converter 3.14159 para a string "3.14159", e só depois gerar código para fazer a concatenação das strings.

4 Exemplos

4.1 Programa correcto

Ao compilar o exemplo referido ao início do documento com a opção `-asm` activa (que mostra os bytecodes gerados em forma textual),

```
/*
   Exemplo de programa em Sol
*/
print 1 + 2 * 3;
print 1 + 2.0 * 3;
print 7 % (1 + 4);
print true and (5 < 3.14159);
print "pi = " + 3.14159;
print "ma" + "ria" == "maria";
```

obtive o seguinte output:

```
Constant pool
    0: 2.0
    1: 3.14159
    2: "pi = "
    3: "ma"
    4: "ria"
```

```
    5: "maria"
Generated code in text format
    0: iconst 1
    1: iconst 2
    2: iconst 3
    3: imult
    4: iadd
    5: iprint
    6: iconst 1
    7: itod
    8: dconst 0
    9: iconst 3
   10: itod
   11: dmult
   12: dadd
   13: dprint
   14: iconst 7
   15: iconst 1
   16: iconst 4
   17: iadd
   18: imod
   19: iprint
   20: tconst
   21: iconst 5
   22: itod
   23: dconst 1
   24: dlt
   25: and
   26: bprint
   27: sconst 2
   28: dconst 1
   29: dtos
   30: sadd
   31: sprint
   32: sconst 3
   33: sconst 4
   34: sadd
   35: sconst 5
   36: seq
   37: bprint
   38: halt
Saving the bytecodes to inputs-t2/in1.tbc
```

E ao correr os bytecodes gerados com a máquina virtual feita no trabalho anterior,

obtém-se:

```
7
7.0
2
false
pi = 3.14159
true
```

4.2 Programa com erros de inconsistência de tipos

O seguinte programa não tem erros de parsing, mas tem erros semânticos (de inconsistência de tipos) e por isso não é compilado para bytecodes.

```
print 1 + 2.0 * 3;
print false or true - 6;
print 1 + true;
print 3 == 3.0;
print "ana" >= "maria";
print 1 + 2 * 400 % (2.0 + 3.1415);
```

Eis o output do compilador:

```
line 2:20 error: operator - is invalid between bool and int
line 3:8 error: operator + is invalid between int and bool
line 5:12 error: operator >= is invalid between string and string
line 6:18 error: operator % is invalid between int and real
inputs-t2/inErros.sol has 4 type checking errors
```

5 Recomendações

1. Começem por especificar a gramática em ANTLR num ficheiro chamado `Sol.g4` e certifiquem-se que a vossa gramática está correcta. Para tal devem testar vários inputs (com e sem erros gramaticais).

2. Após certificarem-se que a gramática está correcta, implementem a deteção de erros semânticos, que no contexto deste trabalho se resumem a erros de inconsistência de tipos. Para tal devem percorrer a árvore sintática e anotar os nós das expressões com informação sobre o seu tipo.

3. Após certificarem-se que não há erros de inconsistência de tipos, podem implementar o gerador de código. Para tal, devem percorrer novamente a árvore sintática e, usando a informação obtida no ponto anterior, gerar instruções adequadas para a máquina virtual T.

4. Não basta testarem estes 2 exemplos que eu ilustrei no enunciado. O vosso trabalho deve ser testado de forma exaustiva para garantir com o máximo de certeza possível que o vosso compilador funciona do modo esperado.

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 22/Abr/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 um ficheiro com o mesmo nome mas substituindo a extensão `.sol` por `.tbc`. O ficheiro de output deverá ser criado na mesma pasta onde estão os ficheiros de input.
- Podem (e devem) correr os bytecodes gerados pela implementação da vossa máquina virtual feita no trabalho anterior, 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.

7 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 25%.