Compiladores, 2023/2024

Trabalho prático, parte 1
– Assembler e Máquina virtual –

Fernando Lobo

1 Introdução

Um assembler é um programa que traduz uma linguagem textual de baixo nível para código máquina ou bytecodes. Por outras palavras, um assembler é um compilador em que a linguagem fonte está muito próxima da linguagem destino.

Neste trabalho pretende-se que faça 2 programas:

- um assembler
- uma máquina virtual capaz de executar/interpretar o código gerado pelo assembler

O assembler recebe como input um programa escrito na linguagem *Tasm*, e produz como output (caso não haja erros) uma sequência de bytecodes para uma máquina virtual chamada T. A especificação da linguagem Tasm e da máquina virtual T é apresentada nas secções 2 e 3.

2 Linguagem Tasm

Tasm é uma abreviatura de *T assembly*, uma linguagem assembly inventada por mim. Por convenção um programa em Tasm tem a extensão `.tasm`, do mesmo modo que um programa em Java tem a extensão `.java`.

Um programa Tasm é uma sequência de uma ou mais instruções, uma por linha. Linhas em branco no ficheiro de input devem ser ignoradas. As linhas de código podem ser anotadas por uma sequência de etiquetas (labels) seguidas do caracter ':'. Uma sequência de etiquetas corresponde a 1 ou mais etiquetas separadas por vírgula.

Independentemente de ser anotada por etiqueta(s), uma linha de código corresponde a uma instrução da máquina virtual T. Uma instrução tem um nome e poderá ter um argumento.

2.1 Exemplo de um programa em Tasm

```
1              galloc 3
2              dconst 3.14159
3              gstore 0
4              iconst 1
5              gstore 2
6   beginLoop: gload 2
7              iconst 3
8              ilt
9              jumpf endLoop
10             gload 0
11             gload 2
12             itod
13             dmult
14             gload 2
15             itod
16             dmult
17             gstore 1
18             sconst "Area de circulo de raio "
19             gload 2
20             itos
21             sadd
22             sconst " = "
23             sadd
24             gload 1
25             dtos
26             sadd
27             sprint
28             gload 2
29             iconst 1
30             iadd
31             gstore 2
32             jump beginLoop
33    endLoop: sconst "Fim"
34             sprint
35             halt
```

2.2 Conjunto de instruções Tasm

De seguida apresenta-se o nome e o tipo de argumento de todas as instruções.

Nome	Argumento
iconst	inteiro, sem sinal
iprint	
iuminus	
iadd	
isub	
imult	
idiv	
imod	
ieq	
ineq	
ilt	
ileq	
itod	
itos	
dconst	inteiro ou real, sem sinal
dprint	
duminus	
dadd	
dsub	
dmult	
ddiv	
deq	
dneq	
dlt	
dleq	
dtos	

Nome	Argumento
sconst	string delimitada por aspas
sprint	
sadd	
seq	
sneq	
tconst	
fconst	
bprint	
beq	
bneq	
and	
or	
not	
btos	
jump	etiqueta
jumpt	etiqueta
jumpf	etiqueta
galloc	inteiro, sem sinal
gload	inteiro, sem sinal
gstore	inteiro, sem sinal
halt	

Como se pode constatar, a maioria das instruções não tem qualquer argumento. Repare também que há instruções com nomes semelhantes e cuja única diferença está na letra inicial, tal como `iconst`, `dconst`, e `sconst`.

O tipo de argumento, caso exista, varia consoante o tipo de instrução, e corresponde a um tipo de token da linguagem.

As regras lexicais para inteiros e reais são idênticos aos que existem na linguagem C. A regra lexical para uma etiqueta é idêntica à regra existente em C para o nome de um identificador.

2.3 Regras semânticas

As regras lexicais e sintáticas da linguagem Tasm já foram explicadas. Passemos agora a descrever algumas verificações semânticas que devem ser verificadas após a análise sintática.

- Um programa Tasm tem forçosamente de ter pelo menos uma instrução `halt`
- Uma etiqueta que é referenciada numa instrução `jump`, `jumpt`, ou `jumpf`, tem de estar forçosamente a anotar uma linha de código algures no programa.
- O nome das etiquetas que anotam linhas de código têm de ser únicos.

3 Máquina virtual T

A máquina virtual T é uma máquina de stack, tal como ilustrado na realização do problema da aula prática P5. Trata-se de uma máquina virtual em tudo semelhante à que foi ilustrada para esse problema prático. A diferença é que tem um conjunto de instruções mais alargado, tem uma zona de memória para poder armazenar variáveis globais, tem uma tabela (*constant pool*) para armazenar constantes que não cabem em 4 bytes (strings e números reais), e o stack de execução passa a poder ter valores de vários tipos, a saber: int, real, bool, string.

Para além destes 4 tipos de dados, a máquina virtual ainda tem um tipo adicional chamado NIL, que significa ausência de valor.

De seguida apresenta-se as instruções da máquina virtual e o seu significado.

Instruções com 1 argumento

OpCode	**Argumento**	**Descrição**
iconst	inteiro n	*int constant*: empilha o valor n no stack
dconst	inteiro n	*real constant*: empilha a constante que está na posição n da *constant pool* (supostamente um valor real), no stack.
sconst	inteiro n	*string constant*: empilha a constante que está na posição n da *constant pool* (supostamente uma string), no stack.
jump	inteiro *addr*	*unconditional jump*: actualiza o *instruction pointer* de modo a que a próxima instrução a ser executada seja aquela que se encontra na posição *addr* do array de instruções.

jumpt	inteiro *addr*	*jump if true*: faz pop do stack. Se o valor for `true`, actualiza o *instruction pointer* de modo a que a próxima instrução a ser executada seja aquela que se encontra na posição *addr* do array de instruções.
jumpf	inteiro *addr*	*jump if false*: faz pop do stack. Se o valor for `false`, actualiza o *instruction pointer* de modo a que a próxima instrução a ser executada seja aquela que se encontra na posição *addr* do array de instruções.
galloc	inteiro n	*global memory allocation*: aloca n posições num array que permite armazenar variáveis globais. Designemos esse array por $Globals$. Essas n posições de memória ficam inicializadas com o valor NIL.
gload	inteiro *addr*	*global load*: empilha $Globals[addr]$ no stack.
gstore	inteiro *addr*	*global store*: faz pop do stack e guarda o valor em $Globals[addr]$

Instruções sem argumentos

OpCode	Descrição
iprint	*int print*: faz pop do operando a, e escreve o seu valor no ecrã.
iuminus	*int unary minus*: faz pop do operando a, e empilha $-a$ no stack. (a e b devem ser valores inteiros)
iadd	*int addition*: faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha $a + b$ no stack.
isub	*int subtraction*: faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha $a - b$ no stack.
imult	*int multiplication*: faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha $a * b$ no stack.
idiv	*int division*: faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha a/b no stack.
imod	*int modulus*: faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o resto da divisão de a por b no stack.
ieq	*int equal*: faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a == b$ no stack.
ineq	*int not equal*: faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a \neq b$ no stack.
ilt	*int less than* faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a < b$ no stack.
ileq	*int less or equal* faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha o valor lógico de $a \leq b$ no stack.

itod	*int to real* converte o valor int que está no topo do stack para um valor real
itos	*int to string* converte o valor int que está no topo do stack para uma string. Ex: 53 é convertido para "53"
dprint	equivalente a iprint, mas para um valor real.
duminus	equivalente a iuminus, mas para um valor real.
dadd	equivalente a iadd, mas para valores do tipo real.
dsub	equivalente a isub, mas para valores do tipo real.
dmult	equivalente a imult, mas para valores do tipo real.
ddiv	equivalente a idiv, mas para valores do tipo real.
deq	equivalente a ieq, mas para valores do tipo real.
dneq	equivalente a ineq, mas para valores do tipo real.
dlt	equivalente a ilt, mas para valores do tipo real.
dleq	equivalente a ileq, mas para valores do tipo real.
dtos	equivalente a itos, mas para um valor real.
sprint	equivalente a iprint, mas para uma string.
sadd	*string concatenation*: faz pop do operando direito b, seguido de pop do operando esquerdo a, e empilha a concatenado com b no stack.
seq	equivalente a ieq, mas para strings.
sneq	equivalente a ineq, mas para strings.
tconst	empilha o valor `true`, do tipo bool, no stack .
fconst	empilha o valor `false`, do tipo bool, no stack .
bprint	equivalente a iprint, mas para um valor do tipo bool.
beq	equivalente a ieq, mas para valores do tipo bool.
bneq	equivalente a ineq, mas para valores do tipo bool.
and	*bool and*: faz pop do operando direito b, seguido de pop do operando esquerdo a (supostamente ambos do tipo bool), e empilha o valor lógico a *and* b no stack.
or	*bool and*: faz pop do operando direito b, seguido de pop do operando esquerdo a (supostamente ambos do tipo bool), e empilha o valor lógico a *or* b no stack.
not	*bool not*: faz pop do operando direito a (supostamente do tipo bool), e empilha o valor lógico *not* a no stack.
btos	equivalente a itos, mas para um valor do tipo bool. Ex: true é convertido para "true".
halt	termina a execução do programa.

4 Recomendações

Para fazer o Assembler:

1. Começem por especificar a gramática em ANTLR num ficheiro chamado `Tasm.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 as verificações semânticas referidas na secção 2.3 deste documento. Para tal devem usar um visitor ou um listener que percorre a árvore de parsing e processa as etiquetas transformando-as em números inteiros que correspondem a linhas de código.

3. Após certificarem-se que não há erros semânticos, podem implementar o gerador de código. Para tal, devem percorrer novamente a árvore de parsing com outro visitor (ou listener) e, usando informação obtida no ponto anterior, gerar instruções para a máquina virtual T.

4. Ao final, devem guardar o código gerado (constant pool e as instruções propriamente ditas) em bytecodes para um ficheiro.

Para fazer a máquina virtual:

1. Começem por ler o ficheiro de input contendo os bytecodes, e descodifiquem a informação para obterem a constant pool e o vosso array de instruções.

2. Uma vez tendo isso, devem executar o código, começando na instrução 0 e terminando após executar a instrução `halt`.

Uma outra abordagem, que talvez até seja mais indicada, é fazerem ambas as coisas de forma incremental. Isto é, façam o vosso Assembler apenas para um conjunto restrito de instruções Tasm, seguido da implementação da máquina virtual capaz de executar essas instruções. Depois, gradualmente vão acrescentando mais instruções até terem o trabalho completo.

5 Condições de realização

O projeto 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 08/Mar/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 `.tasm` que usaram para testar o vosso código
- O vosso assembler deverá chamar-se `tAssembler`. Deverá receber como input um ficheiro com extensão `.tasm` e gerar como output um ficheiro com o mesmo nome mas substituindo a extensão `.tasm` por `.tbc`. O ficheiro de output deverá ser criado na mesma pasta onde estão os ficheiros de input.
- A vossa máquina virtual deverá chamar-se `tVM`
- A gramática em ANTLR deve ter o nome `Tasm.g4`

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

6 Validação e avaliação

Os trabalhos devem ser validados na semana que começa a 08/Mar durante as aulas PL. A validação é obrigatória.

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

A ponderação deste trabalho na nota final é de 30%.