Compiladores

Exame, 23/06/2009

3º ano do MIEIC, FEUP/Universidade do Porto

Duração: 2 horas e 30 minutos

Uma Possível Resolução do Exame de Época Normal

1) [4 valores] Analisadores Gramaticais:

Start → S $
S → a S c
S → B x
B → B b
B → b

a) Indique os conjuntos First(S), First(B), Follow(S), e Follow(B).

First(S) = {a, b}
First(B) = {b}
Follow(S) = {$, c}
Follow(B) = {b, x}

b) Esta gramática pode ser implementada por um analisador sintáctico LL(1)? Justifique a resposta dada.

A gramática tem recursividade à esquerda e por isso não pode ser implementada por analisadores gramaticais LL.

∨

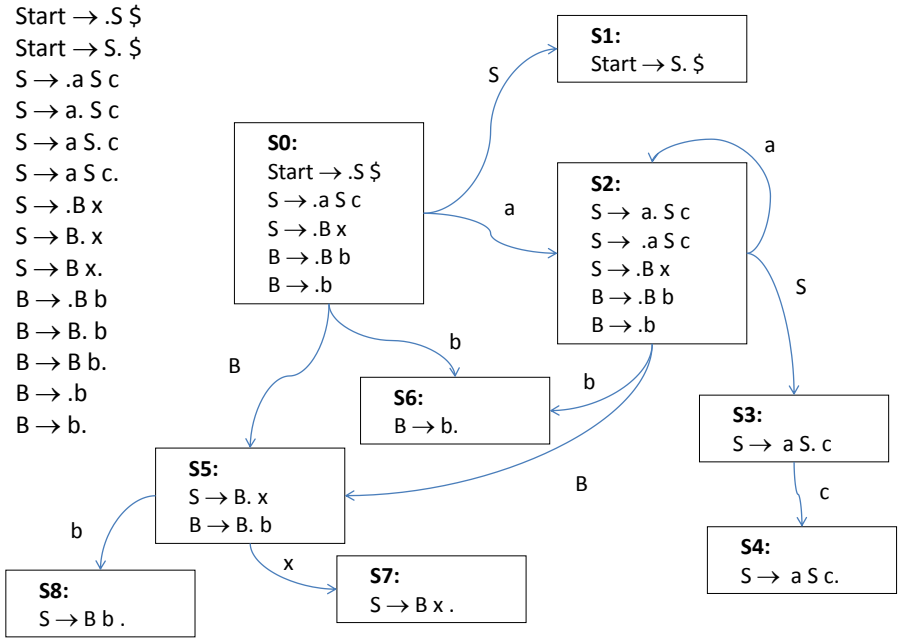
Existem duas produções do símbolo não-terminal B com o mesmo conjunto First, First(B → B b) = First(B → b) = {b}, que invializam a análise com LL(1).

c) Determine o DFA e a tabela do analisador LR(0) para esta gramática.

Itens e DFA para o parser LR(0):

Itens LR(0):

Start → .S $
Start → S. $
S → .a S c
S → a. S c
S → a S. c
S → a S c.
S → .B x
S → B. x
S → B x.
B → .B b
B → B. b
B → B b.
B → .b
B → b.



Identificação das produções:
Start → S $ (1)
S → a S c (2)
S → B x (3)
B → B b (4)
B → b (5)

Tabela do Parser LR(0):

	a	b	x	c	$	S	B
s0	shift S2	shift S6				goto S1	goto S5
s1					accept		
s2	shift S2	shift S6				goto S3	goto S5
s3				shift S4			
s4	red. (2)	red. (2)	red. (2)	red. (2)	red. (2)		
s5		shift S8	shift S7				
s6	red. (5)	red. (5)	red. (5)	red. (5)	red. (5)		
s7	red. (3)	red. (3)	red. (3)	red. (3)	red. (3)		
s8	red. (4)	red. (4)	red. (4)	red. (4)	red. (4)		

d) Esta gramática pode ser considerada uma gramática LR(0)? Porquê?

Pode. A tabela do parser LR(0) não apresenta conflitos redução/redução ou redução/deslocamento.

e) Indique a gramática obtida após a eliminação da recursividade à esquerda.

Start → S $
S → a S c
S → B x
B → b B'

B' → b B'
B' → ε

∨

Start → S $
S → a S c
S → B x
B → b B
B → b

2) [3 valores] Análise Semântica

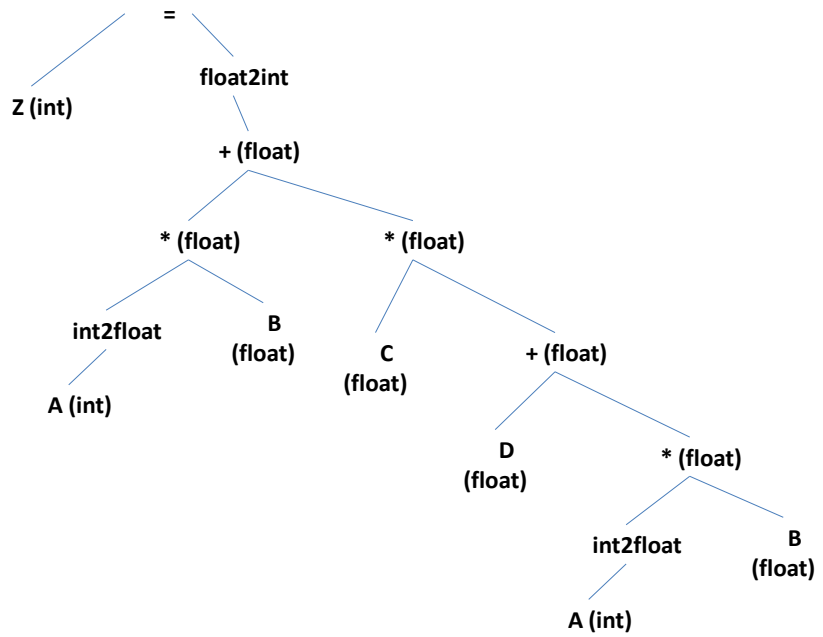
Dado o enunciado Z=A*B+C*(D+A*B); existente numa função de um programa em C e em que os identificadores Z, A, B, C, e D representam variáveis:

a) Indique os aspectos que a análise semântica de um compilador deve verificar aquando da análise da AST que representa a expressão aritmética oriunda do programa em C.

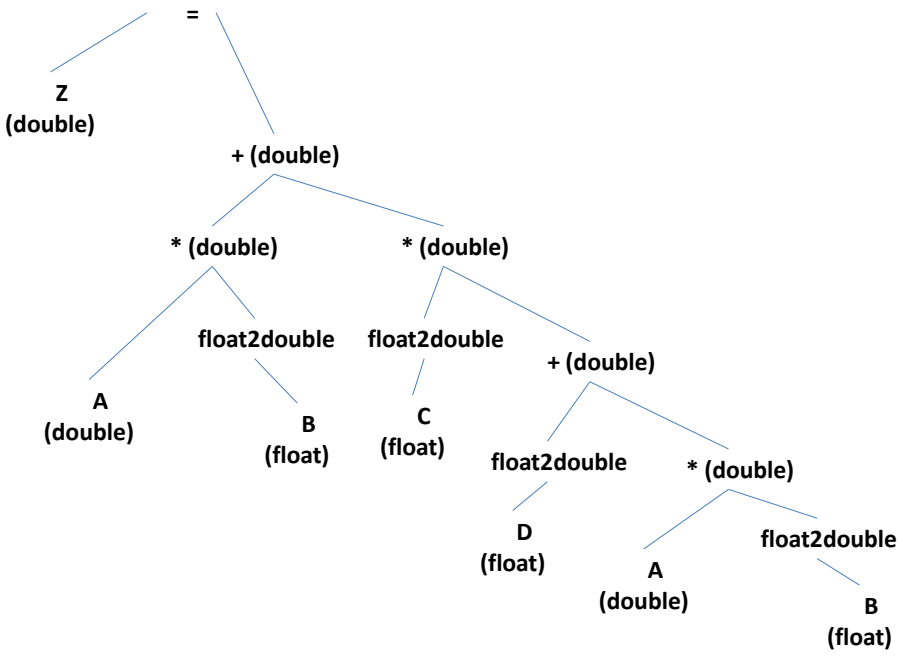
A análise semântica de um compilador deve verificar:
- Se todas as variáveis foram declaradas.
- Quais as declarações das variáveis A, B, C, D e Z no escopo da expressão aritmética.
- Se as variáveis foram declaradas com os tipos compatíveis com a forma como são utilizadas na expressão artimética: int, float, double, long, short, char,… (por exemplo, se A é um array de inteiros não pode ser usado na expressão como A, mas apenas como A[…])
- Quais as conversões a realizar de acordo com os tipos dos operandos e dos resultados
- Quais os tipos das operações a realizar.

b) Supondo que as variáveis B, C, e D são variáveis locais do tipo *float*, indique o resultado da análise semântica em termos do tipo de operações e das conversões necessárias entre tipos a realizar, considerando dois casos: um em que Z e A são do tipo *int*, e o outro em que Z e A são do tipo *double*. Considere que no final da análise semântica é devolvida uma representação intermédia de alto-nível baseada na AST.

a) B, C, e D são variáveis do tipo *float* e Z e A são do tipo *int*:



b) B, C, e D são variáveis do tipo *float* e Z e A são do tipo *double*:



3) [5 valores] Análise de Fluxo de Dados

a) Por que motivo a análise do fluxo de dados (*dataflow analysis*), commummente utilizada para resolver alguns problemas em compiladores, utiliza um algoritmo que itera até encontrar o ponto fixo?

A análise de fluxo de dados é utilizada sempre que é necessário determinar os pontos sobre o fluxo do programa que uma determinada propriedade do programa (definição de uma variável, subexpressão, etc.) pode alcançar. Como um programa pode ter fluxo de controlo ciclico, o fluxo de dados vai sendo construído por passagens sobre a estrutura do programa (sobre o CFG, por exemplo) até que seja atingido um ponto estável (o ponto fixo). A análise de fluxo de dados é feita utilizando a propagação de informação (dados) entre nós adjacentes no CFG. A propagação entre nós do CFG vai sendo determinada iterativamente e tem em conta a possibilidade de fluir por laços recorrentes (uma definição num nó pode atingir usos em nós anteriores devido ao facto de poder fluir pelos laços recorrentes do grafo). Ao atingir o ponto fixo é sinal que a informação que se procurava já percorreu os possíveis fluxos de programa.

b) Sem recorrer a um algoritmo iterativo, como o apresentado para *dataflow analysis*, explique como poderia determinar o tempo de vida das variáveis em trechos de código sem instruções de controlo de fluxo (i.e., em que a sequência de instruções é sempre executada do início até ao fim, sem possibilidade de haver saltos). Indique o pseudo-código desse algoritmo, e socorra-se dos exemplos que considerar relevantes para explicar o funcionamento do mesmo.

Vamos considerar que as n instruções estão armazenadas num vector (1 a n) e que temos para cada instrução os conjuntos de definições e de usos (def e use):

```
for each n
   in[n] ← {}; out[n] ← {}

for i=n; i>=1; i--
     if(i==n)
              out[n] ← live-out(bloco)
     else
              out[n] ← in[n+1]

     in[n] ← use[n] ∪ (out[n] – def [n])  // in[1] = live-in(block)
```

4) [5 valores] Alocação de Registos

O trecho de código seguinte é baseado numa representação intermédia (IR) de baixo-nível. Esta representação assume a existência de uma instrução na máquina alvo para cada instrução da IR. São incluídos os conjuntos de *live-in* e de *live-out* para o trecho de código. Os identificadores h0 e h1 representam constantes.

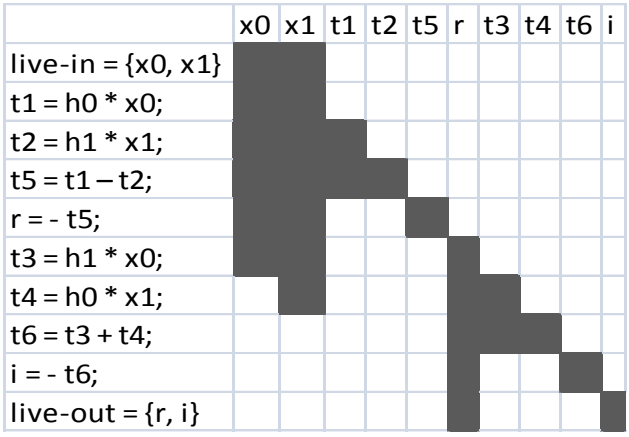
```
live-in = {x0, x1}
t1 = h0 * x0;
t2 = h1 * x1;
t5 = t1 – t2;
r = - t5;
```

t3 = h1 * x0;
t4 = h0 * x1;
t6 = t3 + t4;
i = - t6;
live-out = {r, i}

[2 valores]

a) Indique o tempo de vida de cada variável no trecho de código;

live-in = {x0, x1}
t1 = h0 * x0;
t2 = h1 * x1;
t5 = t1 – t2;
r = - t5;
t3 = h1 * x0;
t4 = h0 * x1;
t6 = t3 + t4;
i = - t6;
live-out = {r, i}



b) Indique, utilizando o algoritmo *Left-Edge*, o número de registos (k) necessários para armazenar todas as variáveis no trecho de código. Indique os passos principais do algoritmo e a alocação de registos resultante considerando os registos R1, R2, ..., Rk.

Passos do algoritmo *left-edge*:

1. Sort segments (live range) by their start time (ascending order)
2. Start by the first segment and try to merge each of the other segments with this one (two segments are merged is they don't overlap)
3. When there is no possibility to merge other segments goto step 2 considering the next segment in the sorted list
4. Number of register = number of columns with segments

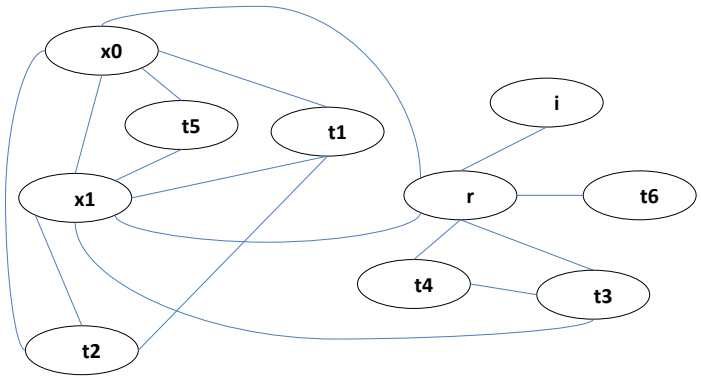
	R1	R2	R3	R4
live-in = {x0, x1}	x0	x1		
t1 = h0 * x0;				
t2 = h1 * x1;			t1	
t5 = t1 – t2;				t2
r = - t5;			t5	
t3 = h1 * x0;			r	
t4 = h0 * x1;	t3			
t6 = t3 + t4;		t4		
i = - t6;	t6			
live-out = {r, i}	i			

Alocação de registos:
R1={x0, t3, t6, i}, R2={x1, t4}, R3={t1, t5, r}, R4={t2}

[3 valores]

c) Utilizando o algoritmo de coloração de grafos para a alocação de registos apresentado nas aulas teóricas, indique a atribuição de variáveis a registos resultante considerando 5 registos: R1, R2, R3, R4, e R5. Apresente a ordem de colocação de nós do grafo de interferências na pilha utilizada pelo algoritmo.

Grafo de interferências:



Possível ordem dos nós do grafo de interferências na pilha: i – t6 – t4 – t3 – r – t5 – x0 – t2 – x1 – t1 (topo da pilha)

Atribuir cores (registos) pela ordem na pilha (a começar pelo topo da pilha)

Possível alocação de registos:
R1={t1, t5, r}, R2={x1, i, t4, t6}, R3={t3, t2}, R4={x0}

d) Indique a alocação de registos e as instruções de *spilling* que teria de adicionar considerando apenas 3 registos R1, R2, e R3, e a existência de

operações de load e de store (ex.: R = M[addr1]; M[addr2] = R; em que M[] representa acessos a memória, addr1 e addr2 representam endereços de memória, e R qualquer um dos 3 registos).

Note-se que não é referido qual o método de alocação de registos a utilizar.

Como precisamos de fazer spilling de pelo menos uma variável podemos escolher a variável com o maior tempo de vida pois é aquela que produz mais interferências. Escolhemos por isso a variável x1.

Para facilitar podemos utilizar a alocação realizada na alínea b). Vamos fazer spilling da variável x1 considerando a alocação obtida na alínea b). As duas figuras seguintes ilustram os passos possíveis.

	R1	R2	R3	R4
live-in = {x0, x1}	x0	x1		
M[addr1] ← x1;				
t1 = h0 * x0;				
t2 = h1 * x1;			t1	
t5 = t1 – t2;				t2
r = - t5;			t5	
t3 = h1 * x0;			r	
x1 ← M[addr1];	t3			
t4 = h0 * x1;		x1		
t6 = t3 + t4;		t4		
i = - t6;	t6			
live-out = {r, i}	i			

	R1	R2	R3	R4
live-in = {x0:R1, x1:R2}	x0	x1		
M[addr1] ← R2;				
R3 = h0 * R1;				
R4 = h1 * R2;			t1	
R3 = R3 – R4;				t2
R3 = - R3;			t5	
R1 = h1 * R1;			r	
x1 ← M[addr1];	t3			
R4 = h0 * R2;		x1		
R1 = R1 + R2;		t4		
R1 = - R1;	t6			
live-out = {r:R3, i:R1}	i			

Resultado final que utiliza apenas 3 registos e que foi obtido pelos passos anteriormente ilustrados:

	R1	R2	R3
live-in = {x0:R1, x1:R2}	x0	x1	
M[addr1] ← R2;			
R3 = h0 * R1;			
R2 = h1 * R2;			t1
R3 = R3 – R2;		t2	
R3 = - R3;			t5
R1 = h1 * R1;			r
x1 ← M[addr1];	t3		
R4 = h0 * R2;		x1	
R1 = R1 + R2;		t4	
R1 = - R1;	t6		
live-out = {r:R3, i:R1}	i		

Note-se que este não é o algoritmo comummente utilizado pelos compiladores. O algoritmo utilizado tem por base a coloração de grafos com os passos indicados nas aulas teóricas. Normalmente é um processo iterativo, pois após spilling é realizada uma nova alocação de registos.

e) Indique se a reordenação das instruções num trecho de código pode influenciar a alocação de registos. Justifique a resposta tendo por base o trecho de código apresentado.

Sim, pode. No exemplo do código dado a reordenação de instruções pode produzir alocações diferentes, mas preserva o número de registos necessários (4).

Por exemplo, ao movermos as instruções t5 = t1 – t2; e r = - t5; para o fim não violamos a funcionalidade do trecho de código, mas alteramos as interferências entre os tempos de vidas das variáveis:

```
live-in = {x0, x1}
t1 = h0 * x0;
t2 = h1 * x1;
t3 = h1 * x0;
t4 = h0 * x1;
t6 = t3 + t4;
i = - t6;
t5 = t1 – t2;
r = - t5;
live-out = {r, i}
```

Neste caso aumentamos o tempo de vida das variáveis t1 e t2, mas diminuimos o tempo de vida das variáveis x1 e r.

Um possível exemplo para ilustrar que a roordenação de instruções pode diminuir o número de registos necessários para armazenar as variáveis é o indicado de seguida:

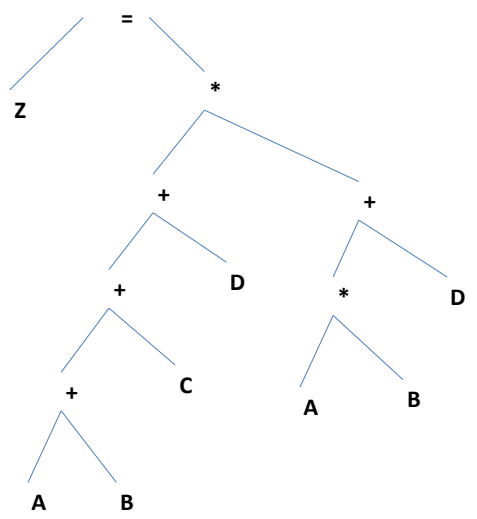
```
live-in = {x0, x1}
t1 = h0 * x0;
t2 = h1 * x1;
t5 = t1 – t2;
r = - t5;
t3 = h1 * x0;
t4 = h0 * x1;
t6 = t3 + t4;
i = - t6;
y = x0+x1;
live-out = {r, i, y}
```

Se movermos a nova instrução para logo a seguir à instrução t3 = h1 * x0; deixa de ser necessário manter vivas x1 e x0 até à última instrução do trecho, e é apenas necessário manter y viva a partir daí.

5) [3 valores] Geração de Código

Considere a expressão aritmética Z=(A+B+C+D)*(A*B+D). Neste caso, Z, A, B, C, e D representam variáveis locais do tipo int:

a) Desenhe uma AST (*Abstract Syntax Tree*) que represente a expressão anterior.



b) Assumindo a seguinte atribuição das variáveis locais da JVM (*Java Virtual Machine*) a cada variável na expressão: Z→1, A→2, B→3, C→4, D→5, indique os passos de um algoritmo para gerar o código para a JVM com o minímo tamanho da pilha de operandos possível e apenas as 5 variáveis locais (considere as instruções: iload <var num>, istore <var num>, iadd, e imul).

Numa primeira fase podemos utilizar uma versão do algoritmo de Sethi-Ullman para rotular cada nó da AST tendo em conta que estamos a lidar com a geração de código JVM:

- No caso da JVM temos de anotar cada folha com uma variável ou constante com 1 (e.g., loads necessitam de armazenar o valor lido na pilha de operandos).
- Para simplificar não vamos considerar variáveis do tipo double ou long. Estas requerem dois níveis da pilha.
- Instruções *dup* duplicam o topo da pilha e por isso devem ser rotuladas com label(n) = label(child) + 1;
- Para instruções store de variáveis locais: label(n) = label(child) - 1;
- Para instruções aritméticas de dois operandos:

```
if label(child1) == label(child2) then
    label(n) = label(child1) + 1;
else
    label(n) = max(label(child1), label(child2))
```

- Para instruções aritméticas de um operando (e.g., *ineg*): label(n) = label(child)
- Para instruções de conversão entre tipos (e.g., i2f, f2i, f2d): label(n) = label(child)

Depois da AST anotada de acordo com o algoritmo de Sethi-Ullman com as modificações explicadas resumidamente pelos passos anteriores, podemos começar a gerar código. Vamos percorrendo a AST começando pela raiz, e geramos por cada nó ou folha a instrução JVM correspondente (note-se que no mesmo nível da AST, é necessário gerar primeiro a instrução correspondente a uma folha antes de gerar a instrução correspondente a uma operação), e

dando prioridade aos nós da AST com rótulo maior sempre que precisamos de descer na AST. No final, o código gerado encontra-se por ordem inversa à que deve ser utilizada.

Nota: na geração de instruções de load e de store para o exemplo dado utilizar-se-ia a alocação: Z→1, A→2, B→3, C→4, e D→5

c) Escreva o código JVM gerado para a expressão utilizando o algoritmo que descreveu na alínea anterior.

Utilizando stack de tamanho 3:

```
Iload 2
Iload 3
Iadd
Iload 4
Iadd
Iload 5
Iadd
Iload 2
Iload 3
Imul
Iload 5
Iadd
Imul
Istore 1
```

É possível gerar código com a stack de tamanho 2 se usarmos a variável local 1 (Z) para guardar um resultado intermédio (note-se que neste caso utilizamos mais duas instruções e precisamos de um algoritmo diferente):

```
Iload 2
Iload 3
Iadd
Iload 4
Iadd
Iload 5
Iadd
istore 1
Iload 2
Iload 3
Imul
Iload 5
Iadd
iload 1
Imul
Istore 1
```