

Relatório Compiladores

Etapa 7

00274705

Andy Ruiz Garramones

Recuperação de erros:

Para a recuperação de erros foram escolhidos vários tipos de erros. A escolha dos erros a serem selecionados foi pensando-se em erros comuns do dia a dia de um programador ou que são mais prováveis a acontecer, seja por falta de atenção ou digitação errada.

A seguir um código contendo exemplo de todos os erros escolhidos enumerados com um *#número*:

```
1  m = int: ;          //#1
2  i = int: 9;
3  iii = int: 0A;
4  vec = int[2]: 0A 0B;
5
6  main () =          //#2
7  {
8      a = 5+          //#3
9      return 5*       //#4
10 };
11
12 foo (x = int) = float
13 {
14     print 5, (a-)    //#5
15     if (TRUE) then
16     { else           //#6
17     {
18
19     }
20     i = vec[2/]      //#7
21     print foo2(5 4)  //#8
22 };
23
24 ii = int: 10        //#9
25
26 foo2 (xx = int, yy = float) float  //#10
27 {
28     read 2          #11
29 }                  //#12
```

Os tipos de erros são explicados a seguir:

1. Ausência de valor em declaração de variável (se esqueceu de colocar o valor atribuído à variável que está sendo inicializada por exemplo)
2. Função sem tipo declarado (ou porque se esqueceu de colocar o tipo)
3. Expressão inválida em uma atribuição
4. Expressão inválida em um comando de return
5. Expressão inválida em comando de print
6. Ausência de } fechando um bloco aberto
7. Expressão inválida como índice para acesso a posição de vetor
8. Ausência de vírgula separando os parâmetros em uma chamada de função
9. Ausência de ponto-e-vírgula (;) em declaração de variável
10. Ausência de = entre parâmetros de uma função e o tipo (se esqueceu do caractere =)
11. Ausência de identificador após comando read (precisa de um identificador logo após o comando read)
12. Ausência de ponto-e-vírgula (;) em declaração de função
13. Embora não apareça no print, se qualquer dos comandos der erro (attribution, readcommand, printcommand, returncommand, flowcommand ou block) que não caiu em nenhum dos caso acima, será tratado e informado como “comando inválido”.

A execução do código mostrado acima gera:

```
Syntax error [line:1]: Expected value in variable declaration
Syntax error [line:9]: Expected valid expression in variable attribution
Syntax error [line:10]: Expected valid expression for return command
Syntax error [line:10]: Expected declared function to have a type
Syntax error [line:14]: Expected valid expression as argument for print
Syntax error [line:16]: Expected } at the end of the block
Syntax error [line:20]: Expected valid expression as index for vector access
Syntax error [line:21]: Expected , to separate arguments in function call
Syntax error [line:22]: Expected valid command
Syntax error [line:28]: Expected identifier after read command
Syntax error [line:29]: Expected valid command
Syntax error [line:29]: Expected ` = ' character after parameters declaration
Syntax error [line:30]: Expected ; after function declaration
Syntax error [line:30]: Expected ; after variable declaration
```

Otimização:

Como técnica de otimização foi implementado o constant folding. Esta otimização consiste em pré-processar as constantes que se encontram em operações para gerar o código assembly com as operações já resolvidas. Por exemplo, uma atribuição de $5 + 2 * 4$ já pode ser resolvida em tempo de compilação e assim gerar o assembly com o resultado, evitando cálculo desnecessário e custoso para o assembly.

Esta otimização foi implementada a nível de AST: após gerar a nossa árvore de sintaxe abstrata foi chamada uma função para otimizar a árvore. Desta forma, todas as operações booleanas e aritméticas foram resolvidas sempre que possível: se houver variáveis ou chamadas de funções não há como pré-processar já agora. Mas se houver parte reduzível, será reduzida e processada.

Abaixo um código fonte simples usado como demonstração:

```
i = int: 9;
a = bool: FALSE;
main () = bool
{
    i = (3 * 4 / (1 + 2) - 1 + (2 * 6) + 1) / ((1 + 2) * 2 - 2)
    a = (5 >= 4 ^ 7 < 5) | 5 <= 2
};
```

Na versão entregue para a etapa 6, sem nenhum tipo de otimização, cada operação na imagem acima será calculada. Isto provoca que a árvore possua muitos mais nodos, que serão traduzidos em mais TAC's e provocarão um código de assembly muito maior. Além do fato que se perderá tempo de processamento em assembly para calcular, de fato, cada uma destas operações.

Versão não otimizada

A seguir a AST gerada por este código fonte na versão sem otimização:

```

ast (AST_DECLIST, 0
| ast (AST_VARDEC, 0
|   | ast (AST_ASSIGNMENT, 0
|   |   | ast (AST_SYMBOL, i
|   |   | ast (AST_SYMBOL, int
|   | ast (AST_SYMBOL, 9
| ast (AST_DECLIST, 0
| ast (AST_VARDEC, 0
|   | ast (AST_ASSIGNMENT, 0
|   |   | ast (AST_SYMBOL, a
|   |   | ast (AST_SYMBOL, bool
|   | ast (AST_SYMBOL, FALSE
| ast (AST_DECLIST, 0
|   | ast (AST_FUNC_VOID_DEC, 0
|   |   | ast (AST_SYMBOL, main
|   |   | ast (AST_SYMBOL, bool
|   |   | ast (AST_BLOCK, 0
|   | ast (AST_LCMDL, 0
|   |   | ast (AST_ATRIBUTION, 0
|   |   |   | ast (AST_SYMBOL, i
|   |   |   | ast (AST_DIV, 0
|   |   |   | ast (AST_PARENTHESIS, 0
|   |   |   |   | ast (AST_ADD, 0
|   |   |   |   |   | ast (AST_ADD, 0
|   |   |   |   |   | ast (AST_SUB, 0
|   |   |   |   |   |   | ast (AST_DIV, 0
|   |   |   |   |   |   |   | ast (AST_MULT, 0
|   |   |   |   |   |   |   |   | ast (AST_SYMBOL, 3
|   |   |   |   |   |   |   |   | ast (AST_SYMBOL, 4
|   |   |   |   |   |   |   |   | ast (AST_PARENTHESIS, 0
|   |   |   |   |   |   |   |   |   | ast (AST_ADD, 0
|   |   |   |   |   |   |   |   |   |   | ast (AST_SYMBOL, 1
|   |   |   |   |   |   |   |   |   |   | ast (AST_SYMBOL, 2
|   |   |   |   |   |   |   | ast (AST_SYMBOL, 1
|   |   |   |   | ast (AST_PARENTHESIS, 0
|   |   |   |   |   | ast (AST_MULT, 0
|   |   |   |   |   |   | ast (AST_SYMBOL, 2
|   |   |   |   |   |   | ast (AST_SYMBOL, 6
|   |   |   | ast (AST_SYMBOL, 1
|   |   | ast (AST_PARENTHESIS, 0
|   |   |   | ast (AST_SUB, 0
|   |   |   |   | ast (AST_MULT, 0
|   |   |   |   |   | ast (AST_PARENTHESIS, 0
|   |   |   |   |   |   | ast (AST_ADD, 0
|   |   |   |   |   |   |   | ast (AST_SYMBOL, 1
|   |   |   |   |   |   |   | ast (AST_SYMBOL, 2
|   |   |   |   | ast (AST_SYMBOL, 2
|   |   | ast (AST_SYMBOL, 2
|   | ast (AST_LCMD, 0
|   |   | ast (AST_ATRIBUTION, 0
|   |   |   | ast (AST_SYMBOL, a
|   |   |   | ast (AST_OR, 0
|   |   |   |   | ast (AST_PARENTHESIS, 0
|   |   |   |   |   | ast (AST_AND, 0
|   |   |   |   |   |   | ast (AST_GE, 0
|   |   |   |   |   |   |   | ast (AST_SYMBOL, 5
|   |   |   |   |   |   |   | ast (AST_SYMBOL, 4
|   |   |   |   |   |   |   | ast (AST_LESSER, 0
|   |   |   |   |   |   |   |   | ast (AST_SYMBOL, 7
|   |   |   |   |   |   |   |   | ast (AST_SYMBOL, 5
|   |   |   | ast (AST_LE, 0
|   |   |   |   | ast (AST_SYMBOL, 5
|   |   |   |   | ast (AST_SYMBOL, 2

```

Na hora da tradução da AST para TAC's isso provoca uma alta quantidade de TAC'S geradas como podemos ver na imagem abaixo:

```
TAC(TAC_BEGINFUN, main, 0, 0);
TAC(TAC_MULT, myWeeirT_emp0, 3, 4);
TAC(TAC_ADD, myWeeirT_emp1, 1, 2);
TAC(TAC_DIV, myWeeirT_emp2, myWeeirT_emp0, myWeeirT_emp1);
TAC(TAC_SUB, myWeeirT_emp3, myWeeirT_emp2, 1);
TAC(TAC_MULT, myWeeirT_emp4, 2, 6);
TAC(TAC_ADD, myWeeirT_emp5, myWeeirT_emp3, myWeeirT_emp4);
TAC(TAC_ADD, myWeeirT_emp6, myWeeirT_emp5, 1);
TAC(TAC_ADD, myWeeirT_emp7, 1, 2);
TAC(TAC_MULT, myWeeirT_emp8, myWeeirT_emp7, 2);
TAC(TAC_SUB, myWeeirT_emp9, myWeeirT_emp8, 2);
TAC(TAC_DIV, myWeeirT_emp10, myWeeirT_emp6, myWeeirT_emp9);
TAC(TAC_COPY, i, myWeeirT_emp10, 0);
TAC(TAC_GE, myWeeirT_emp11, 5, 4);
TAC(TAC_LESSER, myWeeirT_emp12, 7, 5);
TAC(TAC_AND, myWeeirT_emp13, myWeeirT_emp11, myWeeirT_emp12);
TAC(TAC_LE, myWeeirT_emp14, 5, 2);
TAC(TAC_OR, myWeeirT_emp15, myWeeirT_emp13, myWeeirT_emp14);
TAC(TAC_COPY, a, myWeeirT_emp15, 0);
TAC(TAC_ENDFUN, main, 0, 0);
```

O tamanho do assembly gerado é ridiculamente imenso, pois para as aproximadamente 17 operações que temos precisamos gerar o seu cálculo:

```
48 main:
49 pushq %rbp
50 movq %rsp, %rbp
51
52 movl $3, %eax
53 cvtsi2ssl %eax, %xmm0
54 movl $4, %eax
55 cvtsi2ssl %eax, %xmm1
56 mullss %xmm1, %xmm0
57 movss %xmm0, _myWeeirT_emp0(%rip)
58
59 movl $1, %eax
60 cvtsi2ssl %eax, %xmm0
61 movl $2, %eax
62 cvtsi2ssl %eax, %xmm1
63 addss %xmm1, %xmm0
64 movss %xmm0, _myWeeirT_emp1(%rip)
65
66 movss _myWeeirT_emp0(%rip), %xmm0
67 movss _myWeeirT_emp1(%rip), %xmm1
68 divss %xmm1, %xmm0
69 movss %xmm0, _myWeeirT_emp2(%rip)
70
71 movss _myWeeirT_emp2(%rip), %xmm0
72 movl $1, %eax
73 cvtsi2ssl %eax, %xmm1
74 subss %xmm1, %xmm0
75 movss %xmm0, _myWeeirT_emp3(%rip)
76
77 movl $2, %eax
78 cvtsi2ssl %eax, %xmm0
79 movl $6, %eax
80 cvtsi2ssl %eax, %xmm1
81 mullss %xmm1, %xmm0
82 movss %xmm0, _myWeeirT_emp4(%rip)
83
84 movss _myWeeirT_emp3(%rip), %xmm0
85 movss _myWeeirT_emp4(%rip), %xmm1
86 addss %xmm1, %xmm0
87 movss %xmm0, _myWeeirT_emp5(%rip)
88
89 movss _myWeeirT_emp5(%rip), %xmm0
90 movl $1, %eax
91 cvtsi2ssl %eax, %xmm1
92 addss %xmm1, %xmm0
93 movss %xmm0, _myWeeirT_emp6(%rip)
94
95 movl $1, %eax
96 cvtsi2ssl %eax, %xmm0
97 movl $2, %eax
98 cvtsi2ssl %eax, %xmm1
99 addss %xmm1, %xmm0
100 movss %xmm0, _myWeeirT_emp7(%rip)
101
102 movss _myWeeirT_emp7(%rip), %xmm0
103 movl $2, %eax
104 cvtsi2ssl %eax, %xmm1
105 mullss %xmm1, %xmm0
106 movss %xmm0, _myWeeirT_emp8(%rip)
107
108 movss _myWeeirT_emp8(%rip), %xmm0
109 movl $2, %eax
110 cvtsi2ssl %eax, %xmm1
111 subss %xmm1, %xmm0
112 movss %xmm0, _myWeeirT_emp9(%rip)
113
114 movss _myWeeirT_emp9(%rip), %xmm0
115 movss _myWeeirT_emp9(%rip), %xmm1
116 divss %xmm1, %xmm0
117 movss %xmm0, _myWeeirT_emp10(%rip)
118
119 movss _myWeeirT_emp10(%rip), %xmm0
120 cvtsi2ssl %xmm0, %eax
121 movl %eax, _i(%rip)
122
123 movl $5, %eax
124 cvtsi2ssl %eax, %xmm1
125 movl $4, %eax
126 cvtsi2ssl %eax, %xmm0
127 comiss %xmm0, %xmm1
128 jb .L1
129 movl $1, %eax
130 cvtsi2ssl %eax, %xmm0
131 jmp .L2
132
133 .L1:
134 movl $8, %eax
135 cvtsi2ssl %eax, %xmm0
136
137 .L2:
138 movss %xmm0, _myWeeirT_emp11(%rip)
139 movl $7, %eax
140 cvtsi2ssl %eax, %xmm1
141 movl $5, %eax
142 cvtsi2ssl %eax, %xmm0
143 comiss %xmm0, %xmm1
144 jnb .L3
145 movl $8, %eax
146 cvtsi2ssl %eax, %xmm0
147 jmp .L4
148
149 .L3:
150 movl $1, %eax
151 cvtsi2ssl %eax, %xmm0
152
153 .L4:
154 movss %xmm0, _myWeeirT_emp12(%rip)
155 movss _myWeeirT_emp11(%rip), %xmm0
156 pxor %xmm1, %xmm1
157 ucomiss %xmm0, %xmm1
158 jz .L5
159 movss _myWeeirT_emp12(%rip), %xmm0
160 pxor %xmm1, %xmm1
161 ucomiss %xmm0, %xmm1
162 jz .L5
163 movl $1, %eax
164 cvtsi2ssl %eax, %xmm0
165 movss %xmm0, _myWeeirT_emp13(%rip)
166 jmp .L6
167
168 .L5:
169 movl $8, %eax
170 cvtsi2ssl %eax, %xmm0
171 movss %xmm0, _myWeeirT_emp13(%rip)
172
173 .L6:
174 movl $5, %eax
175 cvtsi2ssl %eax, %xmm0
176 movl $2, %eax
177 cvtsi2ssl %eax, %xmm1
178 comiss %xmm0, %xmm1
179 jnb .L7
180 movl $1, %eax
181 cvtsi2ssl %eax, %xmm0
182 jmp .L8
183
184 .L7:
185 movl $8, %eax
186 cvtsi2ssl %eax, %xmm0
187
188 .L8:
189 movss %xmm0, _myWeeirT_emp14(%rip)
190 movss _myWeeirT_emp13(%rip), %xmm0
191 pxor %xmm1, %xmm1
192 ucomiss %xmm0, %xmm1
193 jnz .L9
194 movss _myWeeirT_emp14(%rip), %xmm0
195 pxor %xmm1, %xmm1
196 ucomiss %xmm0, %xmm1
197 jnz .L9
198 movl $8, %eax
199 cvtsi2ssl %eax, %xmm0
200 movss %xmm0, _myWeeirT_emp15(%rip)
201 jmp .L10
202
203 .L9:
204 movl $1, %eax
205 cvtsi2ssl %eax, %xmm0
206 movss %xmm0, _myWeeirT_emp15(%rip)
207
208 .L10:
209 movss _myWeeirT_emp15(%rip), %xmm0
210 cvtsi2ssl %xmm0, %eax
211 movl %eax, _a(%rip)
212
213 .L11:
214 movl $8, %eax
215 cvtsi2ssl %eax, %xmm0
216
217 .L12:
218 movss %xmm0, _myWeeirT_emp11(%rip)
219 movl $7, %eax
```

Com isto, um código extremamente simples de umas poucas linhas acabava gerando um assembly enorme, além de que afetava a performance de certa forma do executável, já que processava em tempo de execução coisa que podia estar já resolvida.

Versão otimizada

Na sua versão otimizada, a AST ficou da seguinte forma:

```
| ast (AST_DECLIST, 0
| | ast (AST_VARDEC, 0
| | | ast (AST_ASSIGNMENT, 0
| | | | ast (AST_SYMBOL, i
| | | | ast (AST_SYMBOL, int
| | | | ast (AST_SYMBOL, 9
| | ast (AST_DECLIST, 0
| | | ast (AST_VARDEC, 0
| | | | ast (AST_ASSIGNMENT, 0
| | | | | ast (AST_SYMBOL, a
| | | | | ast (AST_SYMBOL, bool
| | | | | ast (AST_SYMBOL, FALSE
| | | ast (AST_DECLIST, 0
| | | | ast (AST_FUNC_VOID_DEC, 0
| | | | | ast (AST_SYMBOL, main
| | | | | ast (AST_SYMBOL, bool
| | | | | ast (AST_BLOCK, 0
| | | | | | ast (AST_LCMDL, 0
| | | | | | | ast (AST_ATRIBUTION, 0
| | | | | | | | ast (AST_SYMBOL, i
| | | | | | | | ast (AST_SYMBOL, 04
| | | | | | | ast (AST_LCMD, 0
| | | | | | | | ast (AST_ATRIBUTION, 0
| | | | | | | | | ast (AST_SYMBOL, a
| | | | | | | | | ast (AST_SYMBOL, FALSE
```

Por conseguinte, a lista de TAC's é também pequena pois é proporcional à AST:

```
TAC(TAC_BEGINFUN, main, 0, 0);
TAC(TAC_COPY, i, 04, 0);
TAC(TAC_COPY, a, FALSE, 0);
TAC(TAC_ENDFUN, main, 0, 0);
```

Por último, temos o código assembly. Podemos reparar na que o código é extremamente enxuto e reduzido devido às poucas TAC's geradas.

```
main:
    pushq    %rbp
    movq     %rsp, %rbp

    movl     $4, %eax
    cvtsi2ssl %eax, %xmm0
    cvttss2sil %xmm0, %eax
    movl     %eax, _i(%rip)

    movl     $0, %eax
    cvtsi2ssl %eax, %xmm0
    cvttss2sil %xmm0, %eax
    movl     %eax, _a(%rip)

.L0:
    cvttss2sil %xmm0, %eax
    leave
    ret
```