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 atribuido 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 (atribuition, readcommand, printcommand, returncommand, flowcommand ou block) que não caiu em nenhum dos caso acimas, 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 declarated 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.

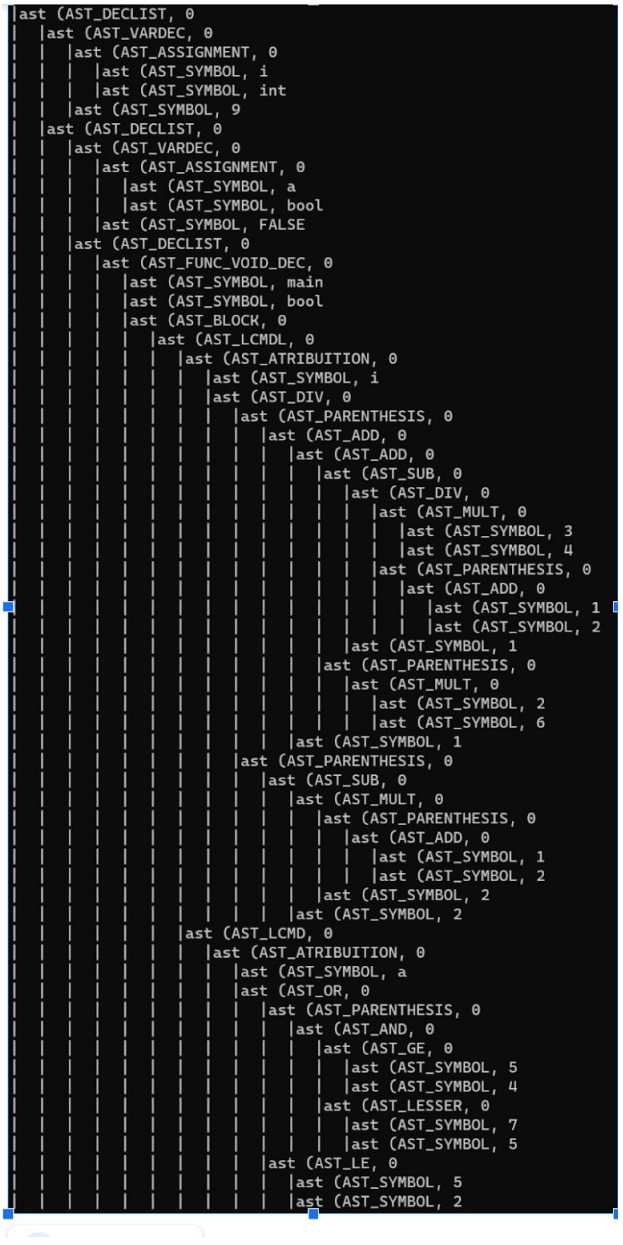
Abaixo um código fonte simples usado como demonstração:

```
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:



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);
```

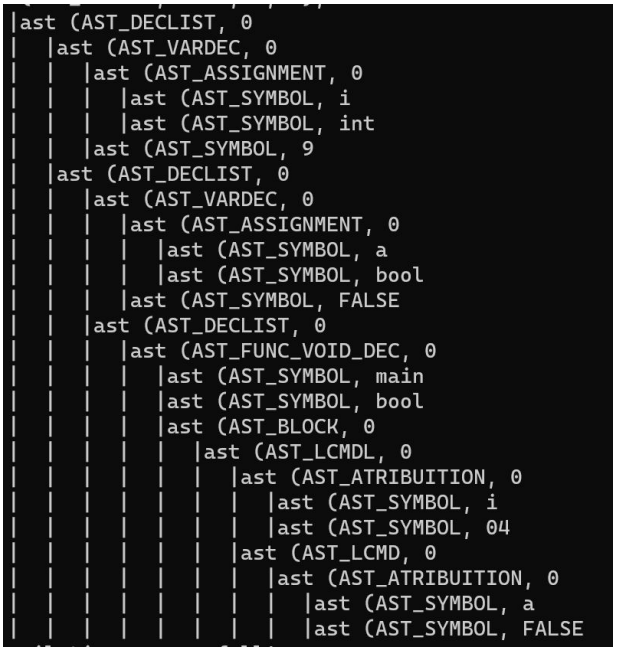
O tamanho do assembly gerado é ridiculamente imenso, pois para as aproximadamente 17 operações que temos precisamos gerar o seu cálculo:

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

    movl    $3, %eax
    cvtsi2ssl   %eax, %xmm0
    movl    $4, %eax
    cvtsi2ssl   %eax, %xmm1
    mulss   %xmm1, %xmm0
    movss   %xmm0, _myWeeirT_emp0(%rip)

    movl    $1, %eax
    cvtsi2ssl   %eax, %xmm0
    movl    $2, %eax
    cvtsi2ssl   %eax, %xmm1
    addss   %xmm1, %xmm0
    movss   %xmm0, _myWeeirT_emp1(%rip)

    movss   _myWeeirT_emp0(%rip), %xmm0
    movss   _myWeeirT_emp1(%rip), %xmm1
    divss   %xmm1, %xmm0
    movss   %xmm0, _myWeeirT_emp2(%rip)

    movss   _myWeeirT_emp2(%rip), %xmm0
    movl    $1, %eax
    cvtsi2ssl   %eax, %xmm1
    subss   %xmm1, %xmm0
    movss   %xmm0, _myWeeirT_emp3(%rip)

    movl    $2, %eax
    cvtsi2ssl   %eax, %xmm0
    movl    $6, %eax
    cvtsi2ssl   %eax, %xmm1
    mulss   %xmm1, %xmm0
    movss   %xmm0, _myWeeirT_emp4(%rip)

    movss   _myWeeirT_emp3(%rip), %xmm0
    movss   _myWeeirT_emp4(%rip), %xmm1
    addss   %xmm1, %xmm0
    movss   %xmm0, _myWeeirT_emp5(%rip)

    movss   _myWeeirT_emp5(%rip), %xmm0
    movl    $1, %eax
    cvtsi2ssl   %eax, %xmm1
    addss   %xmm1, %xmm0
    movss   %xmm0, _myWeeirT_emp6(%rip)

    movl    $1, %eax
    cvtsi2ssl   %eax, %xmm0
    movl    $2, %eax
    cvtsi2ssl   %eax, %xmm1
    addss   %xmm1, %xmm0
    movss   %xmm0, _myWeeirT_emp7(%rip)

    movss   _myWeeirT_emp7(%rip), %xmm0
    movl    $2, %eax
    cvtsi2ssl   %eax, %xmm1
    mulss   %xmm1, %xmm0
    movss   %xmm0, _myWeeirT_emp8(%rip)

    movss   _myWeeirT_emp8(%rip), %xmm0
    movl    $2, %eax
    cvtsi2ssl   %eax, %xmm1
    subss   %xmm1, %xmm0
    movss   %xmm0, _myWeeirT_emp9(%rip)

    movss   _myWeeirT_emp6(%rip), %xmm0
    movss   _myWeeirT_emp9(%rip), %xmm1
    divss   %xmm1, %xmm0
    movss   %xmm0, _myWeeirT_emp10(%rip)

    movss   _myWeeirT_emp10(%rip), %xmm0
    cvttss2sil  %xmm0, %eax
    movl    %eax, _i(%rip)

    movl    $5, %eax
    cvtsi2ssl   %eax, %xmm1
    movl    $4, %eax
    cvtsi2ssl   %eax, %xmm0
    comiss  %xmm0, %xmm1
    jb  .L1
    movl    $1, %eax
    cvtsi2ssl   %eax, %xmm0
    jmp  .L2
.L1:
    movl    $0, %eax
    cvtsi2ssl   %eax, %xmm0
.L2:
    movss   %xmm0, _myWeeirT_emp11(%rip)
    movl    $7, %eax
    cvtsi2ssl   %eax, %xmm1
    movl    $5, %eax
    cvtsi2ssl   %eax, %xmm0
    comiss  %xmm0, %xmm1
    jb  .L3
    movl    $0, %eax
    cvtsi2ssl   %eax, %xmm0
    jmp  .L4
.L3:
    movl    $1, %eax
    cvtsi2ssl   %eax, %xmm0
.L4:
    movss   %xmm0, _myWeeirT_emp12(%rip)
    movss   _myWeeirT_emp11(%rip), %xmm0
    pxor    %xmm1, %xmm1
    ucomiss %xmm0, %xmm1
    jz  .L5
    movss   _myWeeirT_emp12(%rip), %xmm0
    pxor    %xmm1, %xmm1
    ucomiss %xmm0, %xmm1
    jz  .L5
    movl    $1, %eax
    cvtsi2ssl   %eax, %xmm0
    movss   %xmm0, _myWeeirT_emp13(%rip)
    jmp  .L6
.L5:
    movl    $0, %eax
    cvtsi2ssl   %eax, %xmm0
    movss   %xmm0, _myWeeirT_emp13(%rip)
.L6:
    movl    $5, %eax
    cvtsi2ssl   %eax, %xmm0
    movl    $2, %eax
    cvtsi2ssl   %eax, %xmm1
    comiss  %xmm0, %xmm1
    jb  .L7
    movl    $1, %eax
    cvtsi2ssl   %eax, %xmm0
    jmp  .L8
.L7:
    movl    $0, %eax
    cvtsi2ssl   %eax, %xmm0
.L8:
    movss   %xmm0, _myWeeirT_emp14(%rip)
    movss   _myWeeirT_emp13(%rip), %xmm0
    pxor    %xmm1, %xmm1
    ucomiss %xmm0, %xmm1
    jnz    .L9
    movss   _myWeeirT_emp14(%rip), %xmm0
    pxor    %xmm1, %xmm1
    ucomiss %xmm0, %xmm1
    jnz    .L9
    movl    $0, %eax
    cvtsi2ssl   %eax, %xmm0
    movss   %xmm0, _myWeeirT_emp15(%rip)
    jmp  .L10
.L9:
    movl    $1, %eax
    cvtsi2ssl   %eax, %xmm0
    movss   %xmm0, _myWeeirT_emp15(%rip)
.L10:
    movss   _myWeeirT_emp15(%rip), %xmm0
    cvttss2sil  %xmm0, %eax
    movl    %eax, _a(%rip)

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

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.

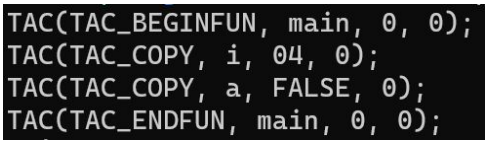
Versão otimizada

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



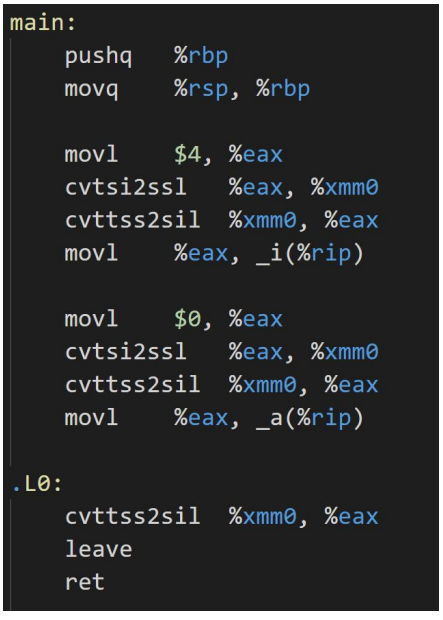
```
|ast (AST_DECLIST, 0
|  |ast (AST_VARDEC, 0
|  |  |ast (AST_ASSIGNMENT, 0
|  |  |  |ast (AST_SYMBOL, i
|  |  |  |ast (AST_SYMBOL, int
|  |  |ast (AST_SYMBOL, 9
|  |ast (AST_DECLIST, 0
|  |  |ast (AST_VARDEC, 0
|  |  |  |ast (AST_ASSIGNMENT, 0
|  |  |  |  |ast (AST_SYMBOL, a
|  |  |  |  |ast (AST_SYMBOL, 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_ATRIBUITION, 0
|  |  |  |  |  |  |  |ast (AST_SYMBOL, i
|  |  |  |  |  |  |  |ast (AST_SYMBOL, 04
|  |  |  |  |  |  |ast (AST_LCMD, 0
|  |  |  |  |  |  |  |ast (AST_ATRIBUITION, 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
```