Relatório Compiladores

Etapa 7 - Recuperação de Erros e Otimização

Recuperação de erros

Para a recuperação de erros foram escolhidos os seguintes tipos de erros:

- ponto-e-vírgula faltando entre as declarações de funções e/ou variáveis globais;
- variável global sem inicialização;
- tamanho do vetor global não ser um literal inteiro;
- declaração dos parâmetros de uma função dentro não estar englobado em parênteses;
- declaração de parâmetros sem identificar o tipo do argumento;
- comando inválido (engloba vários casos, como chamada de função sem atribuir para uma variável, atribuição de expressão incompleta, etc)
- Falta de parênteses na condição do `if/ifelse/while/loop`;
- Expressão incorreta ao indexar vetor;
- Falta de vírgula entre os parâmetros de uma função

No arquivo `test/input.test` pode ser encontrado um arquivo que cause esses erros, com o resultado esperado em `test/expected.test` (que pode ser verificado através do comando `make test`).

Otimização

Foi realizada uma otimização simples, de simplificação de constantes, isso é, caso tenhamos operações binárias entre operadores tanto inteiros quanto booleanos, nós resolvemos a expressão em tempo de compilação.

Por exemplo, imagine a definição de função da figura abaixo (sabendo que o resultado dessa função é sempre TRUE):

```
main() = bool {
    return (7 > 6) ^ (((2 + 1 - 1 + 1) * 5) == 0F) | (5 ≠ 46) ^ (FALSE | (12 ⩽ 42))
};
```

Essa expressão é extremamente complexa, e, sem otimização, gera 63 linhas de assembly para o meu compilador. Executando cada uma das operações acima na CPU, em tempo de execução. Ao otimizarmos para calcularmos essa expressão em tempo de compilação, geramos um código assembly de 7 linhas (4 delas utilizadas para configurar o stack), que pode ser visto na figura abaixo. Assim, se desconsiderarmos as linhas para configurar a stack, temos 59 vs 3 linhas, que dá uma redução de 95% de

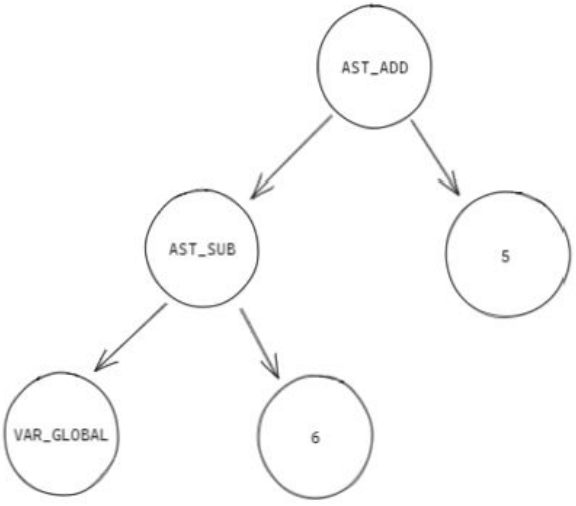
linhas assembly. É claro que o caso é extremo, mas ainda assim, mostra uma melhora no tempo de execução.

```
    .globl  _main
_main:
    # TAC_BEGINFUN
    pushl   %ebp
    movl    %esp, %ebp
    andl    $-16, %esp
    subl    $16, %esp

    # TAC_RETURN
    movl    $0x1, %eax
    leave
    ret
```

A implementação funciona procurando nodos da AST que contenham 2 filhos que possam ser unidos em um só, por já serem constantes inteiras ou booleanas, e satisfaçam a condição de merge. Isso traz um problema com o caso em que tivermos uma expressão como a da figura abaixo à esquerda, que gera uma AST como a da figura abaixo à direita.

```
main() = bool {
    return var_global - 6 + 5
};
```



Esse problema acontece, pois até poderíamos unir o 6 com o 5 mas envolve termos que validar mais do que 1 nível da árvore. Portanto, a otimização feita apenas consegue realizar a união de nodos de operações entre constantes que estejam sendo feitas da esquerda para a direita, mas ao encontrarmos uma constante e/ou uma chamada de função, a otimização para, e não prossegue.