Aluno: Mateus Luiz Salvi Cartão: 00229787
Otimizações

Curto circuito em expressões lógicas:
Quando possuímos expressões lógicas complexas como (A | B) & (C | D), analisamos os dois OR antes de decidir se a expressão AND avalia para verdadeiro ou falso, com curto circuito implementado podemos avaliar toda expressão para falso assim que o primeiro OR é falso, já que a expressão AND não teria mais como ser verdadeira. O mesmo pode ser aplicado para cada um dos OR, assim que o primeiro elemento fosse avaliado para verdadeiro, não existe necessidade de se avaliar o segundo, dado que a expressão já é verdadeira.
Isso poderia ser implementado na geração de código assembly, no AND, ao um dos elementos ser avaliado para falso, damos um jump para depois do bloco ou instrução seguinte, para o OR, ao avaliar o primeiro elemento para verdadeiro, damos um jump para o bloco ou expressão seguinte.

Avaliar e gerar assembly em uma única passagem:
Apesar de mais ortogonal e compreensível, nosso trabalho necessita percorrer a AST diversas vezes para analisar o código de entrada e gerar código assembly. Poderíamos então unir etapas do trabalho, que são feitas independentemente uma da outra, gerando-as em uma única passagem, conhecido como One-Pass compiling, o que pouparia tempo de compilação consideravelmente. Porém, existe um trade-off, analisar e gerar o código em uma única passagem nos impossibilita de implementar diversas otimizações, já que menos informação estará disponível nesta única passagem. Aqui teríamos de escolher entre uma compilação mais rápida ou mais otimizações para uma execução mais rápida. Esta seria uma otimização custosa em trabalho, muitas etapas deverão ser alteradas e possivelmente a gramática também.

Generalizar a Hash Table:
Atualmente nossa tabela Hash possui um tamanho fixo de 996 elementos, já alocados na inicialização da análise, o que leva a dois problemas: não podemos ter mais de 996 elementos e mesmo que não tenhamos todos estes elementos a memória já está alocada, o que ocupa memória desnecessária. O ideal seria alocar espaço em memória para a Hash apenas quando temos uma nova inserção na tabela, aumentando ela à critério, poupando memória e possibilitando a análise de programas muito maiores.

Generalizar a AST:
Nossa AST também sofre do mesmo problema da Hash, temos os 4 filhos de cada nodo sendo alocados em memória mesmo que o nó não possua filhos ou possua menos que 4 filhos. Poderíamos poupar consumo de memória generalizando a árvore para conter um número N de filhos em cada nó, alocando-os apenas quando necessários.

Loop reversal:
Implementar loop reversal nos possibilitaria adicionar outras otimizações, tendo em vista que ela diminuiria o número de dependências em um trecho de código, já que geralmente loops FOR tomam bastante tempo e execução de código em um programa. A técnica consiste em inverter um loop incremental para um loop decremental, subtraindo o índice a

cada iteração até chegar em zero, já que comparações com zero são menos custosas que comparações entre dois números.