Atividade 06 - Expressões Constantes 1 - Compilador

Andrei de Araújo Formiga

10 de março de 2025

Agora já vimos como traduzir expressões com operandos constantes para *assembly* (Atividade 03), como fazer a análise léxica (Atividade 04) e sintática (Atividade 05) da linguagem EC1. A partir da árvore sintática, vimos como interpretar os programas EC1 e determinar seu valor. Nesta atividade vamos reunir o que vimos em todas as atividades anteriores e criar um compilador para a linguagem EC1.

1 A linguagem EC1 (Expressões Constantes 1)

Continuamos com a mesma linguagem EC1 já usada nas Atividades 04 e 05.

Um programa na linguagem EC1 é uma expressão aritmética com operandos constantes e usando as quatro operações. Todas as operações devem ser escritas entre parênteses, então não vamos nos preocupar com precedência de operadores.

Alguns exemplos de programas na linguagem EC1:

```
333
(6 * 7)
(3 + (4 + (11 + 7)))
(33 + (912 * 11))
((427 / 7) + (11 * (231 + 5)))
```

A gramática para a linguagem EC1 é:

```
<programa> ::= <expressao>
<expressao> ::= (<expressao> <operador> <expressao>) | <literal-inteiro>
<operador> ::= + | - | * | /
<literal-inteiro> ::= <digito>+
<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

2 Geração de código

Após realizar a análise léxica e sintática do programa de entrada, a parte que falta para um compilador simples de uma linguagem simples como EC1 é a *geração de código*, que é a etapa que produz o programa de saída na linguagem destino.

Uma forma simples de gerar código é usar um conjunto de modelos (*templates*) para gerar trechos de código para cada tipo de nó da árvore sintática.

Vamos começar a gerar código para alguns programas simples, e tentar chegar a técnicas gerais a partir de exemplos de complexidade crescente.

Para um programa EC1 que é composto apenas por uma constante inteira, já sabemos como gerar o código (como visto na Atividade 02):

```
# programa
42

# codigo gerado
mov $42, %rax
```

O resultado final da expressão deve ser colocado no registrador RAX, como na Atividade 02.

Agora vamos considerar um programa com apenas uma operação, uma soma. Para uma expressão como (7 + 11), sabemos como fazer a tradução para *assembly*, mas vamos pensar em um algoritmo geral para fazer a geração do código.

A raiz da árvore é uma operação binária, então o código gerado para a operação deve combinar, de alguma forma, o código gerado para os operandos esquerdo e direito. Assim como outros processos de travessia da árvore (vimos a impressão e interpretação na Atividade 05), a geração de código é um algoritmo recursivo.

O operando esquerdo da soma (7 + 11) é a constante de valor 7; o operando direito é a constante de valor 11. O código gerado para uma operação constante é um `mov` do valor constante para o registrador RAX. Considerando a tradução dos dois operandos, o modelo incompleto para tradução da soma é:

```
# programa
(7 + 11)

# codigo gerado
mov $7, %rax
# ...
mov $11, %rax
# ...
```

Claro que não podemos apenas juntar as duas instruções `mov`, pois a segunda muda o conteúdo de RAX, eliminando o efeito da primeira instrução. Isso significa que o valor de RAX após o primeiro `mov` deve ser guardado em algum outro lugar, já que o código gerado para o operando direito também vai utilizar o registrador RAX.

Uma possibilidade é copiar o conteúdo de RAX (que veio do operando esquerdo) para outro registrador antes do código para o operando direito. Depois disso, o código pode efetuar a operação, o que resulta no seguinte código:

```
# programa
(7 + 11)

# codigo gerado
mov $7, %rax
mov %rax, %rbx
mov $11, %rax
add %rbx, %rax
```

Esse código é um pouco menos eficiente que o código que seria gerado manualmente, mas funciona para traduzir uma expressão como (7 + 11) automaticamente.

Vamos agora aplicar o mesmo algoritmo para traduzir a expressão (7 + (3 + 8)). O modelo de tradução de uma operaçao binária (até agora) é:

1. inclua o código da tradução do operando esquerdo
2. use `mov` para copiar o conteúdo de RAX para RBX (para não perder o resultado do lado esquerdo)
3. inclua o código da tradução do operando esquerdo
4. realize a operação indicada no nó da árvore

Na expressão (7 + (3 + 8)) temos duas somas: a raiz tem como operando esquerdo a constante 7, e como operando direito a soma 3 + 8; essa soma no operando direito da raiz tem duas constantes como operandos, então vamos usar o modelo acima duas vezes:

```
# programa
(7 + (3 + 8))

## codigo gerado
# operando esquerdo
mov $7, %rax
mov %rax, %rbx
```

```
# operando direito
mov $3, %rax
mov %rax, %rbx     # altera o valor de RBX
mov $8, %rax
add %rbx, %rax
# operacao
add %rbx, %rax
```

Este exemplo nos mostra que o modelo de tradução acima não funciona, pois se sempre usarmos o registrador RBX para guardar o resultado de um operando, esse registrador também terá seu valor sobrescrito se o operando direito for uma operação ao invés de uma constante.

Uma solução seria usar RBX para salvar o primeiro valor de RAX, e depois usar RCX para salvar o segundo valor de RAX. Mas se pensamos sobre o algoritmo de tradução, como saber qual registrador deve ser utilizado para salvar o valor de RAX? O gerador de código pode guardar quais registradores estão disponíveis para guardar valores intermediários. Isso complica o algoritmo de geração de código, e também não resolve tudo.

Pelos exemplos que vimos, podemos inferir que, no pior caso, uma expressão com N operadores vai precisar de N registradores para guardar valores intermediários (fora o RAX); pode ser menos, dependendo da expressão, mas existem expressões que chegam no pior caso (você pode tentar criar expressões de N operadores que usam menos de N registradores).

Isso significa que sempre é possível que a expressão precise de mais registradores do que estão disponíveis no processador.

Esse é um problema comum dos geradores de código: a *alocação de registradores*. Usar registradores sempre é desejável, pois é mais eficiente que outras formas de armazenamento; mas o conjunto de registradores é limitado, é impossível usar registradores para todos os valores necessários. Por isso o gerador de código deve alocar os registradores de forma a gerar o melhor código possível, ou algo próximo disso.

3 Usando a pilha para armazenamento

Uma solução mais simples para a tradução das operações binárias é armazenar os resultados intermediários na pilha. É muito comum usar a pilha para armazenar valores temporários, por exemplo as variáveis locais de uma função, e os valores intermediários de expressões. O limite da pilha depende apenas da quantidade de memória RAM disponível ao processador (que tende a ser muito maior que a capacidade dos registradores) e do sistema de memória virtual do sistema operacional.

A linguagem de máquina da maioria dos processadores inclui instruções para operar uma pilha, geralmente chamada de *pilha do sistema*. No caso da arquitetura x86-64, vamos usar as instruções `PUSH` e `POP`. Ambas as instruções utilizam o registrador RSP, o ponteiro para a pilha (*stack pointer*).

A pilha começa em um endereço alto e cresce na direção dos endereços menores. A instrução `PUSH` decrementa RSP e armazena o valor empilhado no topo da pilha. `POP` faz o contrário: copia o valor apontado por RSP (o topo da pilha) para o local especificado na instrução, e incrementa RSP depois.

Mas qual o valor de RSP? Como inicializar a pilha com valores corretos? Felizmente o programador não precisa se preocupar com isso. Ao criar o processo para executar o programa, o sistema operacional (no caso, o Linux) já cria um segmento para a pilha e inicializa o registrador RSP de forma correta. O sistema operacional também integra o segmento da pilha com o sistema de memória virtual, de forma que o espaço alocado para a pilha possa aumentar caso seja necessário.

4 O esquema de tradução usando a pilha

Com a possibilidade de usar a pilha a qualquer momento no programa, podemos criar um esquema de tradução para operações binárias usando apenas dois registradores, independente do tamanho da expressão. A chave é usar a pilha para guardar o resultado do operando direito, antes de executar o código para o operando esquerdo.

O processo de tradução é:

1. Incluir o código gerado para o operando direito
2. Usar a instrução `push %rax` para salvar o valor do registrador RAX na pilha
3. Incluir o código gerado para o operando esquerdo
4. Desempilhar o resultado no topo da pilha (que é o resultado do operando direito) e colocá-lo em outro registrador (por exemplo RBX)
5. Executar a operação aritmética adequada (soma, multiplicação, etc)

Voltando aos exemplos anteriores, teríamos a seguinte tradução:

```
# programa
(7 + 11)

# codigo gerado
mov $7, %rax     # op. direito
push %rax
mov $11, %rax    # op. esquerdo
pop %rbx
add %rbx, %rax
```

Para que esse esquema funcione, é importante que o código gerado não altere o estado da pilha entre o começo e o final da execução. Ou seja, o estado da pilha ao final da execução do código gerado deve ser igual ao estado da pilha antes da execução do código gerado, mesmo que o código use a pilha no meio.

Para nosso esquema de tradução, a tradução de constantes não usa a pilha; para operações binárias, o esquema sempre empilha um valor na pilha (de RAX) e desempilha um valor da pilha, e portanto a pilha sempre termina na mesma situação que estava no começo do código.

Um detalhe importante é a ordem dos operandos na operação. Para soma (e multiplicação) não faz diferença fazer A+B ou B+A, mas para subtração e divisão a ordem é importante. Dependendo da ordem, o esquema de tradução talvez tenha que ser alterado. Uma forma de mudar a ordem é, ao invés de usar uma instrução `pop %rbx` para recuperar o resultado do operando direito, copiar o RAX após o operando esquerdo para RBX, e usar `pop %rax` para colocar o resultado do operando direito em RAX.

5 Artefato para entrega

O grupo deve entregar o código para o compilador EC1 completo, gerando *assembly* correto para todas as operações, e expressões de qualquer tamanho. Devem ser incluídos testes, de preferência com verificação automática dos resultados. O projeto também deve incluir documentação que explica claramente como usar o compilador.

6 Uma nota sobre otimização

O esquema de código usando a pilha é simples de implementar, mas o código gerado seguindo o esquema não é o melhor possível. Para uma operação simples com dois operandos constantes, o código usa um espaço na pilha e dois registradores; mas é possível calcular a operação usando apenas dois registradores.

A pilha usa a memória RAM do computador, que é muito mais lenta de usar que os registradores (dependendo dos níveis de cache). Para operações mais complexas, o código gerado continua usando apenas dois registradores, e usa mais espaços na pilha. O ideal seria o código usar os registradores ao máximo, e usar a pilha apenas caso não fosse possível fazer tudo nos registradores. Mas, para isso, o esquema de tradução teria que ser mais complexo e fazer mais análises no código; ou o compilador deveria incluir uma etapa de otimização que melhorasse o código depois de gerado.

Veremos como usar esquemas de tradução melhores e como fazer otimização em atividades futuras.

(Na verdade, como todas as expressões na linguagem EC1 são constantes, a expressão inteira pode ser calculada em tempo de compilação e o código gerado seria apenas colocar o resultado da operação no registrador; por exemplo, a expressão `(7 * (78 / (5 + 8)))` poderia

ser traduzida para a única instrução `mov $42, %rax`. Essa otimização é normalmente chamada de *propagação de constantes*.)