Compiladores
Roteiro de Laboratório 07 – Compilador para EZLang

1 Introdução

A tarefa deste laboratório é implementar o *back-end* do compilador para a linguagem EZLang, que gera código *assembly* para uma arquitetura alvo simples. Para facilitar o desenvolvimento deste laboratório, não vamos usar arquiteturas de processadores reais porque elas são muito elaboradas (mesmo as RISC). Ao invés disso, vamos gerar código para a Tiny Machine (TM), descrita na próxima seção. É extremamente desejável que você tenha concluído com sucesso o Laboratório 06, pois a implementação do compilador deste roteiro é uma evolução natural do interpretador do laboratório anterior.

A TM possui um conjunto de instruções bastante restrito, podendo ser considerada uma versão simplificada de uma arquitetura RISC. Por outro lado, também foram inseridos nos *opcodes* da TM instruções complexas, como as que lidam com manipulação de *strings*. Isso foi feito para simplificar o desenvolvimento do laboratório, de forma a evitar o uso de bibliotecas do sistema operacional, como a `libc`. Dessa forma, o estudo neste roteiro fica mais auto-contido. Convém destacar também as muitas similaridades entre as características (semântica) da linguagem de entrada EZLang e o funcionamento (*opcodes*) da TM. Essas similaridades são obviamente intencionais, com o intuito de simplificar a tarefa até os pontos mais fundamentais do problema de geração de código em um compilador. Ao utilizar uma arquitetura simplista como a TM, estamos evitando complexidades adicionais causadas por características específicas de uma ou outra arquitetura alvo, e focando somente nos pontos básicos de implementação de um *back-end*.

2 TM – Tiny Machine

A TM é uma arquitetura simples que pode ser usada como código alvo para exercitarmos a geração de código de um compilador. A TM não existe fisicamente, portanto ela é simulada em *software*. Comece baixando o arquivo `CC_Lab07_src_c.zip` disponibilizado. Esse arquivo contém a implementação do simulador, bem como a parte incompleta do compilador que você deve desenvolver.

A TM possui 1024 posições de memória de instruções (`instr_mem`) e 1024 posições de memória de dados (`data_mem`), além de 32 registradores inteiros (denotados por `i0` até `i31`), 32 registradores de ponto flutuante (`f0 - f31`) e um registrador especial para o contador do programa (PC – *program counter*). O tamanho das memórias e o número de registradores podem ser modificados através de macros definidas no arquivo `instruction.h`.

A arquitetura possui 33 instruções no total, com o número de operandos de uma instrução variando entre 0 e 3. Os *opcodes* de todas as instruções estão definidos através de uma enumeração no arquivo `instruction.h`. Os comentários incluídos no código deste arquivo devem ser suficientes para se entender o funcionamento das instruções. De qualquer forma, seguem algumas explicações adicionais abaixo.

- **Instruções triviais**: `HALT` termina a execução da TM. Todo programa deve ter `HALT` como a última instrução. `NOOP` é a instrução usual que não faz nada além consumir um ciclo de *clock* e avançar o PC.

- **Instruções aritméticas**: as quatro instruções aritméticas possuem variantes inteiras (sufixo `i`) e de ponto flutuante (sufixo `f`), operando sobre os respectivos bancos de registradores. Todos os números possuem sinal e as operações não lidam com *overflow* ou *underflow*, sendo todas realizadas módulo 32 bits. A instrução `WIDf` (*widen to float*) permite a conversão de um inteiro para representação em ponto flutuante. Não há instrução para conversão no outro sentido. A instrução `OROR` é a operação lógica usual do C (`||`).
- **Instruções de comparação**: as duas instruções de comparação possuem variantes para operar sobre inteiros e reais (com os sufixos usuais) e também uma variante para operar sobre *strings* (sufixo `s`). Essas variantes, `EQUs` e `LTHs`, operam diretamente sobre as *strings* armazenadas na tabela de *strings* interna da TM (veja a instrução `SSTR` abaixo). Nesse caso, os registradores operandos contém os índices dessas *strings* na tabela, da mesma forma como foi feito no interpretador do laboratório anterior. Essa última variante para *strings* obviamente não existe em arquiteturas reais, sendo incluída aqui somente para simplificar a manipulação de *strings*, que normalmente requer o uso de chamadas de sistema do SO e alocação dinâmica de memória.
- **Saltos e desvios**: a operação `JUMP` é um salto **absoluto** para algum endereço da memória de instruções. Convém notar que o código *assembly* da TM não permite a criação de rótulos (*labels*), portanto é necessário emitir a instrução com o *endereço* da próxima instrução que se quer executar. Muitas vezes esse endereço não é conhecido no momento que a instrução é gerada, então é necessário fazer o *backpatching* depois (veja os *slides* da Aula 07). As instruções `BOTb` e `BOFb` realizam saltos condicionais segundo o valor Booleano contido no registrador. Fique atento pois esses comandos, ao contrário do `JUMP`, esperam um *offset* **relativo** à instrução atual. Assim, um *offset* positivo avança a execução para alguma instrução à frente (maior endereço) e um *offset* negativo retrocede para alguma instrução anterior.
- **Operações sobre a memória de dados**: as instruções `LDWi` e `STWi`, respectivamente, carregam e armazenam um inteiro na memória de dados (`data_mem`). As variantes para reais operam da mesma forma. Os comandos para carga de constantes em registradores (`LDIi` e `LDIf`) devem ser simples de entender. O único detalhe é que, por questões de limitações do *parser* da TM, a constante real do comando `LDIf` deve ser emitida com a sua representação inteira em uma palavra de 32 bits. Veja os comentários da função `emit_real_val` no arquivo `code.c` para mais explicações.
- **Operações sobre *strings***: conforme explicado anteriormente, para simplificar o desenvolvimento deste laboratório, toda a parte de manipulação de *strings* foi severamente simplificada. Um ponto chave dessa simplificação é a inclusão de uma tabela de *strings* dentro do simulador da TM. Apesar disso, ainda é preciso realizar a conexão entre a tabela de *strings* interna do compilador e a tabela da TM. Isso é feito através das "instruções" `SSTR`. Foi usado o termo entre aspas porque essa não é verdadeiramente uma instrução, mas sim uma diretriz de montagem. A cada instrução `SSTR` lida pela TM, a *string* associada é armazenada na tabela de *strings*. O armazenamento é sequencial, assim cada instrução `SSTR` armazena a *string* no próximo índice da tabela. (É um erro emitir duas instruções `SSTR` com *strings* iguais.) Note que devido à implementação miserável do *parser* da TM, essa instrução não pode ser seguida de um comentário na mesma linha. Considera-se boa prática emitir todas as instruções `SSTR` logo no início do programa. Veja a implementação da função `dump_str_table` no arquivo `code.c` para mais detalhes. A instrução `CATs` concatena as duas *strings* passadas nos registradores `iy` e `iz`, criando uma nova *string* na tabela e armazenando o seu índice no registrador `ix`. As instruções de conversão dos demais tipos para *strings* armazenam o conteúdo do registrador (como uma *string*) na tabela e retornam o seu índice.

- **Simulação das chamadas de SO**: a instrução `CALL` é uma versão simplificada de como são feitas chamadas de sistema (*system calls*) para o *kernel* de um SO. Os códigos das chamadas e as suas funções estão descritos no arquivo `instruction.h`. Foram consideradas somente as funções básicas de entrada e saída no terminal. Essas chamadas envolvem somente um registrador, mas em cenários reais podem ter vários argumentos. (Veja as chamadas da arquitetura x86, se tiver coragem... :P)

Algumas informações adicionais da implementação do simulador da TM:

- Comentários são indicados por `;`. Todos comentários são de linha, não existem comentários de bloco.
- Linhas vazias ou que comecem com um comentário são ignoradas.
- Os *opcodes* devem seguir a capitalização definida no arquivo `instruction.h` e devem sempre aparecer no início da linha (1a. coluna).

Para mais detalhes, veja o código no arquivo `tm.c`. Para compilar o simulador, digite `make tm` no terminal. Isso gera o executável `tmsim`. Um exemplo de um código para TM, incluído no diretório `tests`:

```
; This program reads an integer and output its factorial.

LDIi 0, 0       ; i0 <- 0
LDIi 1, 1       ; i1 <- 1
LDIi 2, 2       ; i2 <- 2
CALL 0, 3       ; i3 <- read int from stdin
LTHi 4, 3, 2    ; i4 <- i3 < i2
LDIi 5, 1       ; i5 <- 1
BOTb 4, 5       ; jump 5 instructions ahead if i3 < i2
MULi 5, 5, 3    ; i5 <- i5 * i3
SUBi 3, 3, 1    ; i3 <- i3 - i1
EQUi 6, 3, 0    ; i6 <- i3 == i0
BOFb 6, -3      ; jump 3 instructions back if i3 == i0
CALL 4, 5       ; stdout <- i5
HALT
```

Procure entender todas as instruções do programa acima, e também o seu funcionamento. Para executar a TM, dispare o simulador como abaixo:

```
$ ./tmsim < tests/fact.tm
*** Instruction Memory:

    0: LDIi 0, 0
    1: LDIi 1, 1
    2: LDIi 2, 2
    3: CALL 0, 3
    4: LTHi 4, 3, 2
    5: LDIi 5, 1
    6: BOTb 4, 5
    7: MULi 5, 5, 3
    8: SUBi 3, 3, 1
    9: EQUi 6, 3, 0
   10: BOFb 6, -3
   11: CALL 4, 5
```

```
   12: HALT

read (int): 5
120
HALTED: Execution finished!
```

Todos os demais detalhes sobre a implementação da TM estão acessíveis no código do arquivo `tm.c`. Se você precisar do *trace* da execução das instruções, retire o comentário do comando na linha 513 do arquivo (`#define TRACE`).

3 Geração de Código

A tarefa deste laboratório é desenvolver o código das funções marcadas com `TODO` no arquivo `code.c`. O funcionamento do gerador de código é muito similar ao funcionamento do interpretador desenvolvido no Laboratório 06: ambos são implementados como um caminhamento em profundidade na AST do programa de entrada. Enquanto o interpretador caminhava pelos nós da árvore executando os comandos, o gerador emite código a medida que visita os nós. Note que, devido à necessidade de *backpatching*, o código emitido não é exibido diretamente em `stdout`. Ao invés disso, o código é temporariamente armazenado em quádruplas, para posterior impressão ao final da compilação (veja os *slides* da Aula 07).

Uma diferença fundamental entre o interpretador e o gerador é a forma como os pedaços do programa transferem dados entre si. No interpretador, todo comando que gerava um valor (expressão) deixava esse valor no topo da pilha para ser usado depois. No entanto, na TM não há uma pilha, portanto o gerador de código deve passar valores pelos registradores. Via de regra, sempre que você precisar de um local para armazenar o resultado de uma operação, você pode realizar uma chamada para `new_int_reg` (ou `new_float_reg`) para obter o número de um novo registrador. Dessa forma, estamos efetivamente trabalhando com temporários (também chamados de registradores virtuais). Uma vez que o código é gerado, deve-se então realizar a *alocação de registradores* reais (conforme descrito no Módulo 08), mas essa tarefa está fora do escopo dessa disciplina.

O caminhamento pela AST segue os mesmos princípios descritos na Seção 2.1 do roteiro do Laboratório 06. Aqui, cada função de emissão de código deve chamar recursivamente a geração para os seus filhos, recebendo como retorno o número do registrador aonde o resultado da expressão do filho vai estar. Por fim, após emitir código correspondente para o nó sendo visitado, a função deve retornar o número do registrador correspondente para a chamada recursiva. Caso o nó não seja uma expressão, utiliza-se um valor de retorno *dummy* (-1) que é descartado depois. O entendimento do funcionamento do interpretador, mais a implementação já pronta para algumas das funções no arquivo `code.c`, deve ser suficiente para você começar. Algumas observações importantes:

- O seu compilador pode terminar a execução ao encontrar o primeiro erro na entrada.
- As mensagens de erros léxicos, sintáticos e semânticos são as mesmas do Laboratório 06 e devem continuar sendo exibidas como antes.
- Os programas de entrada para teste são os mesmos de sempre (`in.zip`). As saídas esperadas desta tarefa estão no arquivo `out07_c.zip`. Os arquivos com extensão `.tm` são a saída do compilador. Já os arquivos `.out` contém o resultado esperado na simulação do programa. Note que o código gerado pelo seu compilador não precisa ficar igual, basta que a saída no simulador esteja correta.