Acadêmico(a):						RA:
Curso: Ciência da Computação		Semestre: 3º		Turma: 3CCOM		Turno:Integral
Professor: Msc. Déverson Rogério Rando		Disciplina: Compiladores				
Data: 07/05/2016	Horário: 08:00	(x) 1º Bim	( ) 2º Bim	( ) Exame	( ) 2° cham	( ) DP/ADAP

Avaliação Bimestral (10.0 Pontos)

Estruturas de Controle

Análise e tradução de construções de controle, como comandos IF e WHILE.

Estratégia

Iremos começar mais uma vez com um "berço" novo, e vamos fazer da mesma forma que já fizemos duas vezes: vamos construir as coisas uma de cada vez. Vamos ainda manter o conceito de um caracter único que nos ajudou tanto até agora. apenas com um "i" no lugar de IF, "w" no lugar de WHILE, etc.

Começando com uma nova cópia do "berço", vamos definir a rotina:

```
/* reconhece e traduz um comando qualquer */
void other()
{
        emit("# %c", getName());
}
```

Agora inclua uma chamada no programa principal:

```
/* PROGRAMA PRINCIPAL */
int main()
{
        init();
        other();

        return 0;
}
```

Execute o programa e veja o que você obtem.

A primeira coisa necessária é a habilidade para tratar de mais de uma linha de comando, uma vez que uma condição de uma só linha é meio limitada. Nós fizemos isto na última parte sobre interpretadores, mas desta vez vamos ser mais formais. Considere a seguinte BNF:

```
        <program> ::= <block> END
        <block> ::= [ <statement> ]*
```

Isto quer dizer que, para os nossos objetivos, um programa é definido como um bloco seguido de um comando END. Um bloco, por sua vez, consiste de zero ou mais comandos. Nós só temos um tipo de comando até aqui.
O que indica o fim do bloco? Simplesmente, qualquer construção que não seja um comando "other". Por enquanto, isto significa apenas o comando END.

Com estas idéias em mente, podemos continuar construindo nosso analisador. O código para o programa é:

```
/* analisa e traduz um programa completo */
void program()
{
        block();
        if (look != 'e')
                expected("End");
        emit("END");
}
```

Note que eu está emitindo um "END". Podemos considerá-lo como uma instrução que interrompe a execução do programa, e faz sentido também, afinal estamos compilando um programa completo.

O código de "block" é:

```
/* analisa e traduz um bloco de comandos */
void block()
{
        while (look != 'e') {
                other();
        }
}
```

Coloque estas rotinas no seu programa. Troque a chamada de `other()` no programa principal para `program()`. Agora teste e veja como funciona. Bem, ainda não é grande coisa, mas estamos chegando lá.

Uma preparação

Antes de começar a definir diversas estruturas de controle, precisamos preparar mais algumas coisas. Antes de mais nada, um aviso: Não as mesmas sintaxes das construções que você está acostumado a encontrar em Pascal ou C. Por exemplo, a sintaxe Pascal para um IF é:

```
IF <condição> THEN <comando>
```

(onde o comando, logicamente, pode ser um bloco)

A versão em C:

```
if ( <condição> ) <comando>
```

Ao invés disso, vamos usar algo que lembra um pouco a linguagem Ada:

```
IF <condição> <bloco> ENDIF
```

Em outras palavras, a construção IF tem um símbolo de terminação específico. Isto evita problemas com "else" perdido como em Pascal e C e também elimina a necessidade de chaves {} ou begin/else. A sintaxe que estou apresentando, na verdade, é da linguagem "KISS" que eu estarei detalhando em capítulos posteriores. As outras construções vão ser um pouco diferentes. Isto não deve ser um problema real pra você. Uma vez que você tenha visto como é feito, vai perceber que não importa muito que sintaxe está sendo usada. Uma vez que ela esteja definida, transformá-la em código é fácil.

Agora, toda construção que tratarmos aqui vai envolver transferência de controle, que significa em assembly, desvios condicionais e incondicionais. Por exemplo, o comando IF abaixo:

```
IF <condição> A ENDIF B
```

deve ser traduzido como:

```
        se NÃO <condição> vá para L
        A
    L:  B
        ...
```

Está claro portanto, que nós vamos precisar de uma rotina a mais para ajudar a lidar com estes desvios. Eu a defini abaixo. A rotina `newLabel()` gera um rótulo único. Isto é feito simplesmente chamando todo rótulo como "Lxx", onde xx é um número começando com zero.

Aqui está:

```
/* gera um novo rótulo único */
int newLabel()
```

```
{
        return labelCount++;
}
```

Precisamos também de um comando para emitir o rótulo:

```
/* emite um rótulo */
int postLabel(int lbl)
{
        printf("L%d:\n", lbl);
}
```

Note que foi adicionada uma nova variável global chamada `lblCount`, então adicione mais uma declaração de variável abaixo da definição de `look`:

```
int labelCount; /* Contador usado pelo gerador de rótulos */
```

Adicione também sua inicialização em `init()`, colocando zero como seu valor.

Neste ponto, vamos ver um novo tipo de notação. Se você comparar a forma do comando IF acima com o código assembly que deve ser produzido você vai notar que há certas ações associada com cada palavra chave no comando:

```
        IF: primeiro, pegar a condição e emitir o código para ela.
            depois, criar um rótulo único e
            emitir um desvio-se-falso para ele.

        ENDIF: emitir o rótulo criado.
```

Estas ações podem ser mostradas de forma concisa, se escrevermos a sintaxe assim:

```
        IF
        <condition>         { <condition>
                            L = newLabel
                            emit(desvio para L) }
        <block>
        ENDIF             { postLabel(L) }
```

Isto é um exemplo de tradução dirigida pela sintaxe. O que está dentro das chaves representa as AÇÕES que devem ser executadas. A parte interessante desta representação é que ela não só mostra o que deve ser

reconhecido, mas também que ações temos que tomar, e em que ordem. Uma vez que temos esta sintaxe, o código praticamente está pronto.
A única coisa que falta fazer é ser mais específico sobre o que é um "desvio se falso".

Estou assumindo que haverá código executado por `condition()` que vai tratar de álgebra booleana e computar algum resultado. Ele deve também alterar os flags de condição correspondentes ao resultado. Agora, a convenção usual para uma variável booleana é que 0000 represente "falso" e qualquer outra coisa (como FFFF, ou 0001) representa "verdadeiro".

No 80x86 os flags condicionais são alterados sempre que qualquer dado é movido ou calculado. Se o dado for 0000 (correspondente a um falso, lembra?) o flag correspondendo a "zero" (ZF - Zero Flag) será alterado para 1. O código para "desvie se zero" é JZ. Então, para os nossos propósito aqui:

```
JZ <=> desvie se falso
JNZ <=> desvie se verdadeiro
```

É de certa forma natural que a maioria dos desvios que vamos encontrar seja da forma JZ... nós vamos "contornar" o código que deveria ser executado se a condição fosse verdadeira.

O comando IF

Com esta pequena explicação, finalmente estamos prontos para começar a codificar o comando IF no nosso analisador. Vamos usar nossa abordagem de um caracter só, "i" para IF, e "e" para ENDIF (como também para END... mas esta duplicidade não vai causar confusão).

O código para `doIf()` (repare que "if" é uma palavra reservada, logo, precisamos usar um identificador diferente) é:

```
/* analisa e traduz um comando IF */
void doIf()
{
        int l;

        match('i');
        l = newLabel();
        condition();
        emit("JZ L%d", l);
        block();
        match('e');
```

```
        postLabel(l);
}
```

Adicione esta rotina ao programa, altere `block()` para se referir a `doIf()` desta forma:

```
/* analisa e traduz um bloco de comandos */
void block()
{
        while (look != 'e') {
                switch (look) {
                  case 'i':
                        doIf();
                        break;
                  default:
                        other();
                        break;
                }
        }
}
```

Note a referência à rotina `condition()`. Eventualmente, vamos escrever uma rotina que possa analisar e traduzir expressões condicionais booleanas. Mas isto é assunto pra um capítulo inteiro. Por enquanto, vamos apenas fazer uma rotina que só emite algum texto. Escreva a seguinte rotina:

```
/* analisa e traduz uma condição */
void condition()
{
        emit("# condition");
}
```

Insira esta rotina e execute o programa. Teste algo assim:

```
        aibece
```

Como você pode ver, o analisador reconhece a construção corretamente e insere o código nos lugares corretos. Agora tente alguns IFs aninhados, como:

```
        aibicedefe
```

Agora que já temos uma idéia geral (e as ferramentas de notação, e também as rotinas `newLabel()` e `postLabel()`), é uma moleza estender o analisador para incluir outras construções. A primeira (e também uma das mais complicadas) é adicionar a cláusula ELSE ao IF. A BNF é:

```
IF <condition> <block> [ ELSE <block> ] ENDIF
```

É um pouco complicado por causa da parte opcional, que não ocorre em outras construções.

A saída correspondente deve ser:

```
        <condition>
        JZ L1
        <block>
        JMP L2
    L1:
        <block>
    L2:
        ...
```

O que nos leva à seguinte tradução dirigida pela sintaxe:

```
IF
<condition>        { L1 = newLabel
                     L2 = newLabel
                     emit(JZ L1) }

<block>
ELSE             { emit(JMP L2)
                     postLabel(L1) }

<block>
ENDIF            { postLabel(L2) }
```

Comparando isso com o caso de um IF sem ELSE nos dá uma dica de como tratar de ambas situações. O código abaixo faz isto. (Note que é usado um "l" para ELSE, já que "e" está sendo usado pra outra coisa.)

```
/* analisa e traduz um comando IF */
void doIf()
{
        int l1, l2;
```

```
        match('i');
        condition();
        l1 = newLabel();
        l2 = l1;
        emit("JZ L%d", l1);
        block();
        if (look == 'l') {
                match('l');
                l2 = newLabel();
                emit("JMP L%d", l2);
                postLabel(l1);
                block();
        }
        match('e');
        postLabel(l2);
}
```

Um analisador/tradutor completo de um IF em 20 linhas de código. Altere também a função `block()`. Troque o teste em `while (look != 'e')` por `look != 'e' && look != 'l'`, ou então o L será tratado por `other()` e nosso IF não vai funcionar.

Faça o teste agora, com alguma coisa assim:

```
aiblcede
```

Funcionou? Agora, pra ter certeza de que está tudo correto com o caso sem ELSE, teste:

```
aibece
```

Agora tente alguns IFs aninhados. Teste tudo o que quiser, incluindo alguns comandos mal formados. Apenas lembre-se que "e" não é um comando válido e sim um terminador.

O comando WHILE

O próximo tipo de comando que deve ser fácil, já que temos o processo assimilado. A sintaxe para o WHILE é:

```
WHILE <condition> <block> ENDWHILE
```

Agora, considere que o WHILE deve ser traduzido como:

```
L1:
    <condition>
    JZ L2
    <block>
    JMP L1
L2:
```

Como antes, comparar as duas representações nos dá uma idéia de que ações são necessárias em cada ponto.

```
WHILE           { L1 = newLabel
                  L2 = newLabel
                  postLabel(L1) }
<condition>       { emit(JZ L2) }
<block>
ENDWHILE        { emit(JMP L1)
                  postLabel(L2) }
```

O código segue diretamente a sintaxe:

```
/* analisa e traduz um comando WHILE */
void doWhile()
{
        int l1, l2;

        match('w');
        l1 = newLabel();
        l2 = newLabel();
        postLabel(l1);
        condition();
        emit("JZ L%d", l2);
        block();
        match('e');
        emit("JMP L%d", l1);
        postLabel(l2);
}
```

Não confundam `doWhile` com o comando `do ... while(<cond>);` da linguagem C. O "do" foi acrescentado ao nome da rotina pra não confundí-lo com a palavra chave `while` de C.

Como temos um comando novo, temos que adicionar a chamada à rotina `block()`:

```
/* analisa e traduz um bloco de comandos */
void block()
{
        while (look != 'e' && look != 'l') {
                switch (look) {
                   case 'i':
                        doIf();
                        break;
                   case 'w':
a                        doWhile();
                        break;
                   default:
                        other();
                        break;
                }
        }
}
```

Nenhuma outra mudança é necessária por enquanto.

Awbece

Tudo o que temos a fazer pra acomodar a nova construção é trabalhar na tradução dirigida pela sintaxe dela. O código praticamente sai diretamente de lá, e quase não afeta outras rotinas.

ATIVIDADES:
FAZER O TRADUTOR PARA OS BLOCOS A SEGUIR
O comando LOOP

É um laço de repetição infinito

A sintaxe é simples:

```
LOOP <block> ENDLOOP
```

e a tradução dirigida pela sintaxe:

```
LOOP    { L = newLabel
          postLabel(L) }
<block>
ENDLOOP { emit(JMP L) }
```

```
/* analisa e traduz um comando LOOP */
```

Repeat-Until

Construção extraída diretamente do Pascal. A sintaxe é:

```
REPEAT <block> UNTIL <condition>
```

A tradução dirigida fica assim:

```
REPEAT          { L = newLabel
                  postLabel(L) }
<block>
UNTIL
<condition>       { emit(JZ L) }
```

```
/* analisa e traduz um REPEAT-UNTIL*/
```

O laço FOR

Sintaxe do BASIC:

```
FOR <ident> = <expr1> TO <expr2> <block> ENDFOR
```

O código traduzido para o processador 80x86:

```
        <ident>           ;pega o nome do contador do laço
        <expr1>           ;pega o valor inicial
        DEC AX            ;pré-decrementa
        MOV [<ident>], AX ;salva o valor do contador
        <expr2>           ;pega o limite superior
        PUSH AX           ;salva na pilha
    L1:
        MOV AX, [<ident>] ;coloca em AX
        INC AX            ;incrementa o contador
        MOV [<ident>], AX ;salva o novo valor
        POP BX            ;pega limite superior...
        PUSH BX           ;...mas devolve na pilha
        CMP AX, BX        ;compara contador com limite superior
        JG L2             ;termina se contador > limite superior
        <block>
        JMP L1            ;próximo passo
    L2:
        POP AX            ;retira limite superior da pilha
```

```
/* analisa e traduz um comando FOR*/
```

Como não temos expressões para este analisador, use a rotina `condition()`:

```
void expression()
{
        emit("# EXPR");
}
```

O Comando DO

Laço que deve ser executado um certo número de vezes. O 80x86 tem uma instrução que decrementa o contador e desvia se não for zero.

A sintaxe e a tradução:

```
        DO
        <expr>    { expression
                  emit(MOV CX, AX)
                  L = newLabel
                  postLabel(L)
                  emit(PUSH CX)
```

```
        <block>
        ENDDO   { emit(POP CX)
                  emit(LOOP L) }


/* analisa e traduz um comando DO */
```