CiCADA

Compilador em C para ADA

Parte 2

Uso:

Para usar, basta compilar:

```
make
```

E correr com o nome do ficheiro de input como argumento:

```
./cicada input.txt
```

Ficheiros criados/incluidos no Trabalho:

- cicada.c - *ficheiro principal que lê um input, chama as funções para atuar sobre o input, e faz output de um ficheiro .asm (assembly em MIPS) e também de vários prints para o stdout*
- st.h - *definição da tabela de símbolos (lista ligada simples)*
- st.c - *implementação da tabela de símbolos (e print da tabela)*
- ic.h - *definição do código intermédio*
- ic.c - *implementação do código intermédio*
- mc.h - *definição da função de imprimir código MIPS*
- mc.c - *implementação da função de imprimir código MIPS*

Exemplo:

Começamos com código em Ada:

```
procedure Test is
    x : Integer;
begin
    Get_Line(x);
    if x >= 42 and x % 2 = 0 then
        x := x + 1;
        Put_Line(x);
    else
        x := x - 1;
        Put_Line(x);
    end if;
end Test;
```

Fazemos o parse do texto com Flex e Bison, criando a Abstract Syntax Tree (AST):

```
PROCEDURE(Test) IS(
    DCLR_SIMPLE(
        Integer(x)
    )
BEGIN(
    FUNCTION(
        Get_Line(
            ID(x)
        )
    )
    IF(
        LOGIC_AND(
            GREATER_OR_EQUAL(
                ID(x)
                INT(42)
            )
            EQUAL(
                MOD(
                    ID(x)
                    INT(2)
                )
                INT(0)
            )
        )
    ) THEN(
        ASSIGN(
            ID(x)
            ARITHMETIC_EXPRESSION(
                PLUS(
                    ID(x)
                    INT(1)
                )
            )
        )
        FUNCTION(
            Put_Line(
                ID(x)
            )
        )
    )
    ELSE(
        ASSIGN(
            ID(x)
            ARITHMETIC_EXPRESSION(
                MINUS(
                    ID(x)
                    INT(1)
                )
            )
        )
        FUNCTION(
            Put_Line(
                ID(x)
            )
        )
    ) END_IF
)) END_Test
```

Percorremos a AST à procura de variáveis e colocámo-las numa lista, a Symbol Table (ST):

```
ID: x, TYPE: Integer
```

Usando a AST e a ST gera-se Código Intermédio (IC), neste trabalho um código de 3 endereços:

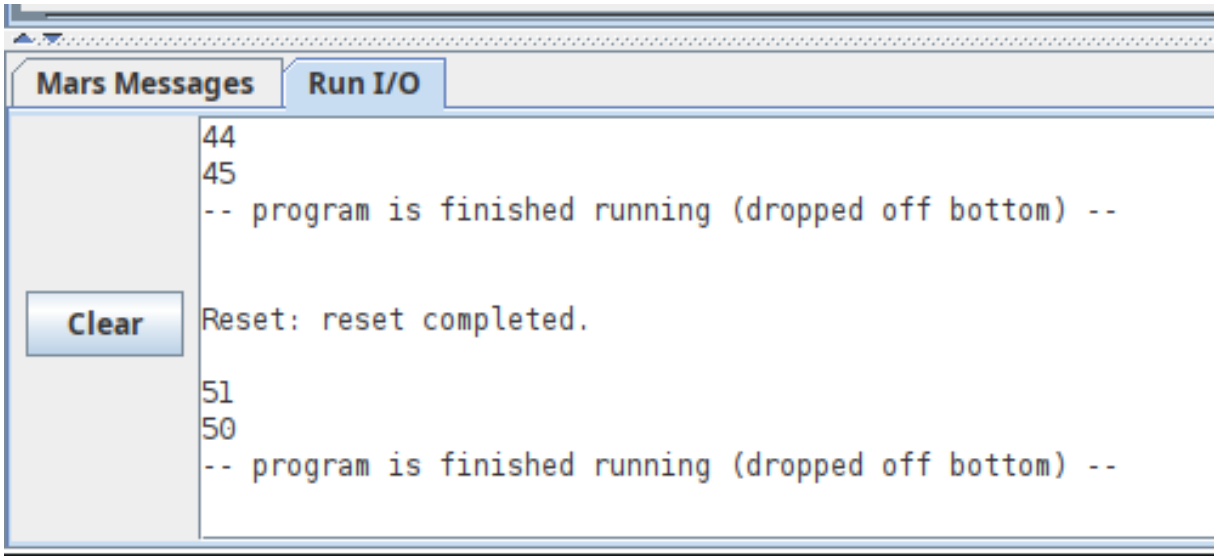
```
GET x
t0 <- x
t1 := 42
COND t0 >= t1 label3 label1
LABEL label3
t2 <- x
t3 := 2
t0 := t2 % t3
t1 := 0
COND t0 == t1 label0 label1
LABEL label0
t0 <- x
t1 <- x
t2 := 1
t0 := t1 + t2
t0 -> x
t0 <- x
PUT t0
JUMP label2
LABEL label1
t0 <- x
t1 <- x
t2 := 1
t0 := t1 - t2
t0 -> x
t0 <- x
PUT t0
LABEL label2
```

Por fim, e como o IC é já muito parecido com o MIPS, é só traduzi-lo para MIPS. Sem esquecer de percorrer a ST uma última vez para declarar todas as variáveis:

```
.data
x: .word 0

.text
main:
    li $v0, 5
    syscall
    sw $v0, x
    lw $8, x
    li $9, 42
    bge $8, $9, label3
    j label1
label3:
    lw $10, x
    li $11, 2
    rem $8, $10, $11
    li $9, 0
    beq $8, $9, label0
    j label1
label0:
    lw $8, x
    lw $9, x
    li $10, 1
    add $8, $9, $10
    sw $8, x
    lw $8, x
    li $v0, 1
    move $a0, $8
    syscall
    j label2
label1:
    lw $8, x
    lw $9, x
    li $10, 1
    sub $8, $9, $10
    sw $8, x
    lw $8, x
    li $v0, 1
    move $a0, $8
    syscall
label2:
```

Este código pode ser corrido no simulador de MIPS MARS:



Com input de 44, devolve 45. Com input de 51, devolve 50.

Estruturas usadas:

Tabela de Símbolos (ST):

A Tabela de Símbolos é representada por uma lista de st_node.

Cada st_node tem:

- char* id - *nome da variável*
- st_type type - *tipo da variável, definido na AST*
- st_node* next - *apontador para o próximo nó*

A lista é uma lista ligada simples e, como na nossa implementação sem âmbitos a ordem das variáveis não interessa, podemos manter apenas a cabeça da lista e ir acrescentando os nós ao início da lista.

Código Intermédio (IC):

O Código Intermédio é uma lista de ic_node.

Cada ic_node tem:

- Instr* instr - *instrução em código de 3 endereços*
- ic_node* next - *apontador para o próximo nó*

A estrutura de uma Instrução em Código de 3 Endereços é:

- Opcode opcode - *Opcode já muito similar ao MIPS (ADD, LABEL, etc.)*
- Addr arg1, arg2, arg3, arg4 - *Argumentos. A nossa implementação usa no máximo 4 argumentos: nas instruções com condições precisamos de guardar os dois operandos da comparação e as duas etiquetas que representam o caminho verdadeiro ou falso depois da comparação.*

A lista é uma lista ligada simples, mas agora precisámos de manter a cabeça (para percorrer a lista) e a cauda, para adicionar nós ao fim da lista em tempo constante (já que as instruções precisam de estar ordenadas).

Código MIPS (MC):

A geração de código MIPS não carece de nenhuma estrutura auxiliar - basta percorrer a lista de Código Intermédio e ir imprimindo as Instruções traduzidas para formato MIPS.

Para usar os nomes da variáveis "em cru" no Código MIPS, é preciso percorrer a Tabela de Símbolos e declarar, no cabeçalho de MIPS, os nomes das variáveis e reservar espaço correspondente ao tipo. No nosso trabalho, aceitamos variáveis de tipo Integer, e estamos a reservar um espaço *.word*. Estamos também a atribuir um valor por defeito de 0. Assim, os nossos Integer são palavras de 32 bits e são inicializados a zero.

Os registos são representados pelos seus valores inteiros e não pelos nomes. Isto porque $t0 é equivalente a $8, mas enquanto que os registos temporários podem ser representados com $8 até $25, seria necessário representar $t0…$t7, $s0…$s7, $t8, $t9 (o que é completamente exequível mas, para este trabalho, desnecessário).

Notas Finais:

Apesar de considerarmos o trabalho completo para efeitos de entrega, há ainda muitas características que não foram incluídas e seriam interessantes de explorar:

- Incluir âmbitos e fazer alocação de registos.
- Implementar funções. A implementação de Get_Line e Put_Line foi feita diretamente no código MIPS (syscall 1 e syscall 5).
- Extender os tipos suportados, de momento só o tipo Integer é suportado.
- Várias optimizações com vários graus de dificuldade de implementação.

O desenho da Abstract Syntax Tree foi limitado pela nossa falta de conhecimento da matéria futura (como não podia deixar de ser). Nesta segunda parte foi necessário reescrever algumas partes da AST, nomeadamente a forma de lidar com Expressões Booleanas (Condições). O resultado é uma estrutra de AST que carece de uma reescrita para ser mais compreensível e fácil de ler. Em particular, o slide 47 da aula 10 mostra um desenho de Expressões e Condições que seria o objetivo final de uma eventual reescrita.

Conversando entre nós, declarámos que este trabalho foi dos mais interessantes do nosso percurso académico. Permite (como o professor salientou numa das aulas) apreender de forma holística o funcionamento de um programa de computador e, assim, o funcionamento de um computador.