Compiladores — Folha laboratorial 1

Pedro Vasconcelos, DCC/FCUP

Setembro 2022

Interpretador para programas sequênciais (C)

Pretende-se escrever um interpretador em C para programas sequenciais com atribuições e expressões aritméticas.

Os programas são constituidos por *expressões* e *comandos* ("*statements*"). As expressões podem ser números, variáveis e operações aritméticas sobre inteiros. Os comandos podem ser atribuições, incrementos ou sequências de outros comandos.

Dois exemplos (usando sintaxe concreta da linguagem C e com os valores finais das variáveis anotados em comentários):

```
// Exemplo 1
a = 3; b = 2; a = a*b;       // a: 6, b: 2
// Exemplo 2
a = 1; a++; b = a*2;         // a: 2, b: 4
```

Estes programas não têm decisões, ciclos nem funções; logo terminam sempre (eventualmente com um erro de execução como uma divisão por zero).

Sintaxe abstrata

Para facilitar o processamento é conveniente converter a sintaxe concreta acima numa *árvore de sintaxe abstrata* (AST); esta conversão é efetuada pelos analisadores lexical e sintático (*lexer* e *parser*) que vamos estudar mais tarde.

Nos exercícios seguintes vamos assumir que temos já programas em sintaxe abstrata, ou seja, como valores de estruturas ligada para comandos `Stm` e expressões `Exp`. As declarações em C são:

```
typedef enum { PLUS, MINUS, TIMES, DIV }
  BinOp;

typedef enum { IDEXP, NUMEXP, OPEXP }
  ExpType;

struct _Exp {
  ExpType exp_t;     // etiqueta
  union {            // alternativas
    char *ident;     // IDEXP
    int num;         // NUMEXP
```

```
    struct {        // OPEXP
      struct _Exp *left;
      BinOp op;
      struct _Exp *right;
    } opexp;
  } fields;
};

typedef struct _Exp *Exp;

typedef enum { ASSIGNSTM, INCRSTM, COMPOUNDSTM }
  StmType;

struct _Stm {
  StmType stm_t;     // etiqueta
  union {            // alternativas
    struct {
      char *ident;
      Exp exp;
    } assign;        // ASSIGNSTM
    char *incr;      // INCRSTM
    struct {
      struct _Stm *fst;
      struct _Stm *snd;
    } compound;      // COMPOUNDSTM
  } fields;
};

typedef struct _Stm *Stm;
```

O ficheiro `ast.c` inclui funções auxiliares para construir árvores sintáticas. O exemplo 1 acima pode ser construido da seguinte forma:

```
Stm exemplo1
    = mk_compound(mk_assign("a", mk_opexp(mk_numexp(5), PLUS, mk_numexp(3))),
                  mk_assign("b", mk_opexp(mk_idexp("a"), MINUS, mk_numexp(2))));
```

O esqueleto do código fornecido num repositório Git que contém os seguintes ficheiros:

`ast.{c,h}` definições de tipos para sintaxe abstracta e funções auxilares para construtores

`interpreter.c` declarações das funções pedidas (para completar).

`table.{ch}` definições de tipos para tabela de associações entre identificadores e valores

`tests.c` módulo principal com definições de alguns de casos de teste para as funções pedidas.

Além deste módulos tem também um ficheiro `Makefile` para automatizar a compilação:

```
$ cd src
$ make
$ ./runtests
```

Inicialmente todos os testes falham porque falta implementar as funções necessárias (exercícios seguintes).

Exercício 1: Testar identificadores

Escreva duas funções recursivas para testar se um identificadores ocorre numa expressão ou num comando:

```
int ids_in_exp(Exp exp, char *ident);
int ids_in_stm(Stm stm, char *ident);   // resultados 0 ou 1
```

Note que a função `ids_in_stm` deve chamar a função `ids_in_exp` porque comandos podem conter expressões.

Exercício 2: Interpretador funcional

Escreva duas funções recursivas para interpretar comandos e expressões:

```
int interpret_exp(Exp exp, Table tbl);
Table interpret_stm(Stm stm, Table tbl);
```

Vamos representar uma *tabela de associações* de variáveis a inteiros pelo tipo `Table` que é uma lista ligada de entradas `Entry` (ver ficheiro `table.c`).

A função `interpret_stm` recebe um comando e tabela e retorna a tabela modificada.

A função `interpret_exp` recebe uma expressão e uma tabela e retorna o valor inteiro da expressão; as variáveis nunca são modificadas pela avaliação da expressão.

Sugestões: use as funções auxiliares de `table.c` para procurar o valor (se existir) associado a um identificador, e para atualizar e acrescentar uma entrada.