Projeto de Compiladores

Análise Semântica e Geração de Código

Bruno Zabot — 202302069

Dezembro de 2025

Conteúdo

1 Introdução **2**

2 Tabela de Símbolos **2**
2.1 Estrutura de Dados 2
2.2 Operações Principais 2

3 Análise Semântica **2**
3.1 Sistema de Tipos 3

4 Código de Três Endereços (TAC) **3**
4.1 Funções Principais 3
4.2 Tradução de Expressões 3
4.3 Tradução de Estruturas de Controlo 4

5 Geração de Código MIPS **4**
5.1 Funções Principais 5
5.2 Gestão de Memória 5
5.3 Tradução de Operações 5
5.4 Registos Utilizados 6

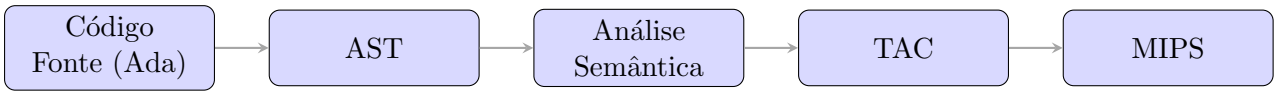
6 Execução **6**

1 Introdução

Este relatório documenta a segunda parte do projeto de compiladores, que estende a implementação anterior (análise léxica, sintática e construção da AST) com as seguintes funcionalidades:

- **Tabela de Símbolos:** Estrutura para armazenar informações sobre variáveis. (symbol_table.{c, h})
- **Análise Semântica:** Verificação de tipos e validação do uso de variáveis. (semantics.{c, h})
- **Código de Três Endereços (TAC):** Representação intermediária do programa. (tac.{c, h})
- **Geração de Código MIPS:** Tradução para assembly executável. (mips.{c, h})
- **interpreter.{c, h} e codegen.{c, h}:** auxiliares responsáveis por chamar os arquivos principais.

O fluxo completo de compilação segue a seguinte sequência:



2 Tabela de Símbolos

A tabela de símbolos armazena informações sobre os identificadores do programa, incluindo nome, tipo e localização em memória.

2.1 Estrutura de Dados

A implementação utiliza uma **hash table** para resolução de colisões, garantindo inserção e busca eficientes:

2.2 Operações Principais

- `create_symbol_table(size)`: Cria tabela com tamanho especificado.
- `insert_symbol(table, name, type, offset)`: Insere ou atualiza símbolo.
- `lookup_symbol(table, name)`: Procura símbolo pelo nome.
- `free_symbol_table(table)`: Libera memória alocada.

A função hash utilizada é a **djb2**.

3 Análise Semântica

A análise semântica verifica a correção do programa além da estrutura sintática.

3.1 Sistema de Tipos

Operação	Tipos Aceitos	Resultado
Aritmética	int × int	int
Aritmética	float × float	float
Aritmética	int × float	float
Relacional	numérico × numérico	bool
Lógica	bool × bool	bool

4 Código de Três Endereços (TAC)

O TAC é uma representação intermediária onde cada instrução possui no máximo três operandos, simplificando a tradução para código de máquina.

4.1 Funções Principais

O arquivo `tac.c` disponibiliza funções para criar cada tipo de instrução TAC:

- `tac_assign_int(dest, val)`: Cria atribuição de constante inteira (`dest = val`).
- `tac_assign_var(dest, src)`: Cria atribuição entre variáveis (`dest = src`).
- `tac_bin_op(dest, src1, op, src2)`: Cria operação binária (`dest = src1 op src2`).
- `tac_label(label)`: Cria um label para saltos.
- `tac_jump(label)`: Cria salto incondicional (`goto label`).
- `tac_cond_jump(src1, relop, src2, label)`: Cria salto condicional (`if src1 relop src2 goto label`).
- `tac_print(src)`: Cria instrução de impressão.
- `tac_read(dest)`: Cria instrução de leitura.

Adicionalmente, `newTemp()` gera nomes únicos para variáveis temporárias (t0, t1, ...) e `newLabel()` gera labels únicos (L0, L1, ...).

4.2 Tradução de Expressões

Expressões complexas são decompostas usando temporários. A função `transExp` percorre recursivamente a AST e gera instruções TAC utilizando as funções acima:

```
1 static void transExp(Expr *expr, char *dest) {
2     switch (expr->kind) {
3         case E_INTEGER:
4             append_tac(tac_list, tac_assign_int(dest, expr->
    attr.value));
5             break;
6
7         case E_OPERATION: {
```

```
8               char t1[16], t2[16];
9               sprintf(t1, "t%d", newTemp());
10              sprintf(t2, "t%d", newTemp());
11
12              // Chamada recursiva para os operandos
13              transExp(expr->attr.op.left, t1);
14              transExp(expr->attr.op.right, t2);
15
16              // Geracao do codigo intermediario
17              append_tac(tac_list, tac_bin_op(
18                  dest,
19                  t1,
20                  expr->attr.op. operator,
21                  t2
22              ));
23              break;
24          }
25      }
26  }
```

Listing 1: Tradução de Expressões

4.3 Tradução de Estruturas de Controlo

A função `transCmd` traduz comandos da AST para TAC. Para estruturas de controlo como `while`, são utilizados labels e saltos:

```
1  case E_WHILE: {
2      char l_start[16], l_body[16], l_end[16];
3      sprintf(l_start, "L%d", newLabel());
4      sprintf(l_body, "L%d", newLabel());
5      sprintf(l_end, "L%d", newLabel());
6
7      append_tac(tac_list, tac_label(l_start));
8      transCond(cmd->attr.while_loop.condition, l_body, l_end);
9
10     append_tac(tac_list, tac_label(l_body));
11     transCmd(cmd->attr.while_loop.body);
12     append_tac(tac_list, tac_jump(l_start));
13
14     append_tac(tac_list, tac_label(l_end));
15     break;
16 }
```

Listing 2: Tradução do While

A função `transCond` traduz expressões booleanas, gerando saltos condicionais para os labels de verdadeiro ou falso.

5 Geração de Código MIPS

O gerador traduz instruções TAC para assembly MIPS executável.

5.1 Funções Principais

O arquivo `mips.c` implementa a tradução de TAC para MIPS:

- `generate_mips_from_tac(list)`: Função principal que percorre a lista de instruções TAC e gera o código MIPS correspondente, incluindo prólogo e epílogo do programa.
- `get_offset(name)`: Gerencia a alocação de variáveis na stack, retornando o offset de uma variável (cria nova entrada se não existir).
- `load_mips(reg, var)`: Gera instrução `lw` para carregar variável da memória para registo.
- `store_mips(reg, var)`: Gera instrução `sw` para armazenar registo na memória.

5.2 Gestão de Memória

Variáveis são alocadas na stack usando o frame pointer (`$fp`). Cada nova variável recebe um offset decrementado de 4 bytes:

```
1 static void load_mips(char *reg, char *var) {
2     int off = get_offset(var);
3     printf("    lw %s, %d($fp)\n", reg, off);
4 }
5
6 static void store_mips(char *reg, char *var) {
7     int off = get_offset(var);
8     printf("    sw %s, %d($fp)\n", reg, off);
9 }
```

Listing 3: Acesso a Variáveis

5.3 Tradução de Operações

Cada tipo de instrução TAC é mapeado para sequências de instruções MIPS:

```
1 // Operacoes aritmeticas
2 case TAC_BIN_OP:
3     load_mips("$t0", curr->src1);
4     load_mips("$t1", curr->src2);
5     switch(curr->op) {
6         case PLUS:     printf("    add $t0, $t0, $t1\n");
    break;
7         case MINUS:    printf("    sub $t0, $t0, $t1\n");
    break;
8         case TIMES:   printf("    mul $t0, $t0, $t1\n"); break
    ;
9         case DIVIDES: printf("    div $t0, $t0, $t1\n");
10                      printf("    mflo $t0\n"); break;
11     }
12     store_mips("$t0", curr->dest);
13     break;
14
```

```
// Saltos condicionais
case TAC_COND_JUMP:
    load_mips("$t0", curr->src1);
    load_mips("$t1", curr->src2);
    switch(curr->op) {
        case EQUALS:  printf("    beq $t0, $t1, %s\n", curr->dest); break;
        case LESS:    printf("    blt $t0, $t1, %s\n", curr->dest); break;
        case GREATER:  printf("    bgt $t0, $t1, %s\n", curr->dest); break;
    }
    break;
```

Listing 4: Operações Aritméticas e Saltos

5.4 Registos Utilizados

Registo	Utilização
`$t0, $t1`	Temporários para operações
`$fp`	Frame pointer (base da stack)
`$sp`	Stack pointer
`$a0`	Argumento para syscalls
`$v0`	Código do syscall / retorno

6 Execução

```
# Compilar o projeto
make

# Executar
./interpreter < programa.ada

# Ou

# Rodar os testes
make test
```