



FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Compiladores Parte 1

DCC-FCUP 2025/26

Gabriel Adriano Ramalho Almeida Carlos

n.º 202306755

Gonçalo de Carvalho Fernandes Branquinho Mota

n.º 202306501

Objetivo do Trabalho

Este projeto implementa as fases da análise léxica (*scanner*) e análise sintática (*parser*) para um subconjunto simples da linguagem de programação **Ada**. O parser, para além da análise sintática, constrói ainda uma **Abstract Syntax Tree (AST)** que representa o programa Ada fornecido como *input*.

O subconjunto da linguagem Ada inclui:

- Um procedimento principal chamado **Main**;
- **Expressões**: aritméticas, booleanas, funções **Put_Line** e **Get_Line**;
- **Comandos**: atribuições, expressões condicionais (**if-then-else**), ciclos **while**.

Decisões de Implementação

Análise Léxica (Lexer.x)

- Todos os identificadores e keywords são convertidos para minúsculas para respeitar o **case-insensitivity**;
- A função **removerAspas** remove as aspas duplas dos literais de strings;

Análise Sintática (Parser.y)

- O parser respeita a gramática do subconjunto de Ada, de forma a garantir que a estrutura do programa é sempre da forma **procedure Main ... is ... begin ... end Main;**
- Declarações podem ser compostas (**DeclComp**) ou vazias (**EmptyDecl**);
- Comandos executáveis podem ser compostos (**ExecComp**) ou vazios (**EmptyComp**);
- A precedência dos operadores, é garantida através de diferentes níveis na gramática (**OrExp**, **AndExp**, **RelExp**, **AddExp**, **MultExp**, **PowExp**, **UnaryExp**, **Factor**);
- O operador unário (-) é convertido numa subtração **Sub (IntLit 0) Factor**.

Estrutura da AST

A AST é representada pelos seguintes tipos de dados:

- **Prog**: representa o programa completo (declarações + comandos);
- **Decl**: representa declarações de variáveis;

- **DeclVar**: lista de nomes de variáveis declaradas;
- **Type**: tipos de dados (**Integer**, **Boolean**, **Float**, **String**);
- **Exec**: comandos executáveis (**Assign**, **IfThenElse**, **WhileLoop**, **PutLine**, **GetLine**);
- **Exp**: expressões (literais, variáveis, operações aritméticas, lógicas, relacionais e de concatenação).

Construção da AST

A AST é construída de forma hierárquica:

- A raiz é o nó **Prog**, que contém as declarações e comandos;
- Expressões são representadas de forma recursiva;
- Simplificamos as fases posteriores do compilador através do agrupamento dos 6 nós (`if $2 then $4 else $6`), resultando em (**IfThenElse** `$2 $4 $6`);
- Convertemos `>` em `<`, e `>=` em `<=`, tendo então a necessidade de implementar apenas 2 operadores.

Tabela de Símbolos

A tabela de símbolos associa nomes a tipos e inclui as funções típicas:

- `empty` (faz return ao estado inicial da tabela);,
- `bind` (cria um novo binding na tabela, ou seja, um par nome-tipo);,
- `enter` e `exit` (para poder entrar e sair de um âmbito, guardam o estado anterior e restauram-no, respetivamente);,
- `lookup` (para que se possa encontrar o registo mais recente de um dado nome - faz return ao tipo, ou a um tipo inválido no caso de não existir; além disso, foram implementadas duas versões desta função: uma que atua durante a construção da tabela, e uma que atua no resultado da tabela).,

Os âmbitos são estáticos e o programa (main) faz return à tabela do âmbito geral, à pilha do estado da tabela antes de entrar em cada âmbito (vazia no fim, sempre) e a uma lista que guarda os vários estados da tabela que correspondem à tabela quando sai de um âmbito local, e a ordem desta lista é a mesma que a do código: a mais à esquerda na lista é o âmbito local que fecha mais cedo no programa, e mais à direita é a do que fecha mais tarde. Por fim, o Main permite

fazer o display destes valores de várias formas, e a implementação de tabela de símbolos está preparada para lidar com âmbitos de formas mais pertinentes. Neste caso, como só verificamos os tipos, os efeitos dos âmbitos não se notam no resultado final da tabela).

Como Compilar e Executar

No diretório raíz do projeto, executar o seguinte script:

```
./scripts/testExamples.sh
```

No fim da execução, pode encontrar o output correspondente a cada exemplo, na pasta “examples”

Exemplos de Input/Output

Exemplo 1:

Input:

```
procedure Main is
    x : Integer := 10;
    y : Integer;
begin
    y := x + 5;
    Put_Line(y);
end Main;
```

Output (AST):

```
Prog
├── DeclComp
│   ├── DeclInit
│   │   ├── DeclVarLast "x"
│   │   └── TypeInteger
│   └── IntLit 10
└── DeclNonInit
    ├── DeclVarLast "y"
    └── TypeInteger
└── ExecComp
    ├── Assign
    │   ├── "y"
    │   └── Add
    │       ├── Var "x"
    │       └── IntLit 5
    └── PutLine
        └── Var "y"
```

Tabela de Símbolos:

```
gabriel@gabitspc:~/School/Compiladores/Projeto$ ghc Main.hs -o symboltable
[3 of 4] Compiling SymbolTable      ( SymbolTable.hs, SymbolTable.o )
[4 of 4] Compiling Main            ( Main.hs, Main.o )
Linking symboltable ...
gabriel@gabitspc:~/School/Compiladores/Projeto$ ./symboltable exemplo1.adb 3
([("y",TypeIntegerST),("x",TypeIntegerST)],([],[]))
gabriel@gabitspc:~/School/Compiladores/Projeto$ ./symboltable exemplo1.adb 1
[("y",TypeIntegerST),("x",TypeIntegerST)]
gabriel@gabitspc:~/School/Compiladores/Projeto$ ./symboltable exemplo1.adb 2
"A quantidade de \226mbitos \233 (escolha um n\250mero de 0 a at\233 ao total de \226mbitos - 1, caso contr\225rio mostra todos): 0"
[]
```

Exemplo 2:

Input:

Output (AST):

```
Prog
├── DeclInit
│   ├── DeclVarLast "score"
│   └── TypeInteger
│       └── IntLit 75
└── IfThenElse
    ├── Le
```

```

1 procedure Main is
2   score : Integer := 75;
3 begin
4   if score ≥ 60 then
5     Put_Line("Aprovado");
6   else
7     if score ≥ 50 then
8       Put_Line("Suficiente");
9     else
10      Put_Line("Reprovado");
11    end if;
12  end if;
13 end Main;

```

Exemplo 3:

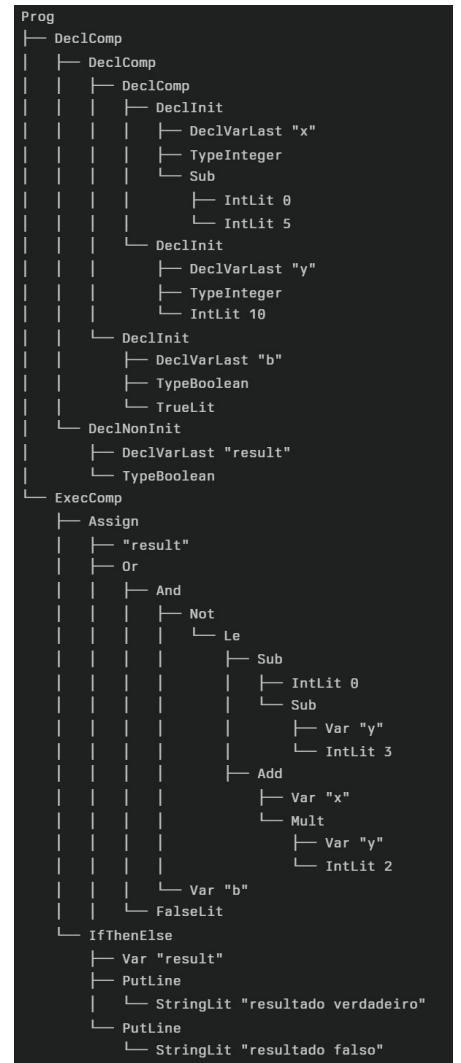
Input:

```

1 procedure mAin is
2   x : Integer := -5;
3   y : Integer := 10;
4   b : Boolean := TRUE;
5   result : Boolean;
6 begin
7   result := not (x + y * 2 ≥ - (y - 3)) and b or false;
8
9   if result then
10    Put_Line("Resultado verdadeiro");
11   else
12    Put_Line("Resultado falso");
13  end if;
14 end mAin;

```

Output (AST):



Exemplo 4:

Input:

```

gabriel@gabispc:~/School/Compiladores/Projeto$ cat exemplo4.ad
procedure Main is
  x : Integer;
begin
  Put_Line("A resposta local é: ");
  declare
    y : Integer := 2;
    z : Integer := 3;
    begin
      x := (y + z);
      Put_Line(x);
    end;
  Put_Line("A resposta pós local é: ");
  Put_Line(x);
end Main;

```

Tabela de Símbolos:

```
gabriel@gabispc:~/School/Compiladores/Projeto$ ./symboltable exemplo4.adb 1
[("x",TypeIntegerST)]
gabriel@gabispc:~/School/Compiladores/Projeto$ ./symboltable exemplo4.adb 2
"A quantidade de \226mbitos \233 (escolha um \n\250mero de 0 a at\233 ao total de \226mbitos - 1, caso contr\225rio mostra todos): 1"
0
[[("z",TypeIntegerST),("y",TypeIntegerST),("x",TypeIntegerST)]]
gabriel@gabispc:~/School/Compiladores/Projeto$ ./symboltable exemplo4.adb 2
"A quantidade de \226mbitos \233 (escolha um \n\250mero de 0 a at\233 ao total de \226mbitos - 1, caso contr\225rio mostra todos): 1"
1
[[("z",TypeIntegerST),("y",TypeIntegerST),("x",TypeIntegerST)]]
gabriel@gabispc:~/School/Compiladores/Projeto$ ./symboltable exemplo4.adb 3
([("x",TypeIntegerST),[],[],[[("z",TypeIntegerST),("y",TypeIntegerST),("x",TypeIntegerST)]]))
```

Exemplo 5:

Input:

```
gabriel@gabispc:~/School/Compiladores/Projeto$ cat exemplo5.adb
procedure Main is
    x : Integer;
begin
    Put_Line("A resposta local \233: ");
    declare
        y : Integer := 2;
        z : Integer := 3;
    begin
        x := (y + z);
        declare
            a : Integer := 1;
            b : Integer;
            begin
                end;
            Put_Line(x);
        end;
    Put_Line("A resposta p\243s local \233: ");
    Put_Line(x);
end Main;
```

Tabela de Símbolos:

```
gabriel@gabispc:~/School/Compiladores/Projeto$ ./symboltable exemplo5.adb 1
[("x",TypeIntegerST)]
gabriel@gabispc:~/School/Compiladores/Projeto$ ./symboltable exemplo5.adb 2
"A quantidade de \226mbitos \233 (escolha um \n\250mero de 0 a at\233 ao total de \226mbitos - 1, caso contr\225rio mostra todos): 2"
1
[[("b",TypeIntegerST),("a",TypeIntegerST),("z",TypeIntegerST),("y",TypeIntegerST),("x",TypeIntegerST)]]
gabriel@gabispc:~/School/Compiladores/Projeto$ ./symboltable exemplo5.adb 2
-A quantidade de \226mbitos \233 (escolha um \n\250mero de 0 a at\233 ao total de \226mbitos - 1, caso contr\225rio mostra todos): 2"
-1
[[("z",TypeIntegerST),("y",TypeIntegerST),("x",TypeIntegerST),("b",TypeIntegerST),("a",TypeIntegerST),("y",TypeIntegerST),("x",TypeIntegerST)]]
gabriel@gabispc:~/School/Compiladores/Projeto$ ./symboltable exemplo5.adb 3
([("x",TypeIntegerST),[],[],[[("z",TypeIntegerST),("y",TypeIntegerST),("x",TypeIntegerST),("b",TypeIntegerST),("a",TypeIntegerST),("y",TypeIntegerST),("x",TypeIntegerST)]]))
```

Segunda Parte

Código Intermédio

Gera código intermediário (representação de três endereços) a partir da AST:

- Instruções essenciais: MOVE, MOVEI, OP, LABEL, JUMP, COND
- Suporte de operações: aritméticas, lógicas, comparações, concatenação
- Rastreamento de escopos para variáveis locais
- Short Circuit
- Reutilização de temporários

Dead Code Elimination (a partir de Liveness Analysis)

Elimina código que não produz efeitos, aliviando a carga do processamento do código. Também quase acabámos de implementar Available Assignments. A análise de liveness não está ativada por default, mas pode ser ativada de assim se quiser (ver o Main.hs, descomentar a linha relevante).

MemoryAllocator

Aloca registradores MIPS e espaço na stack para cada temporário e variável:

- Registradores inteiros: \$t0-\$t9 (temporários), \$s0-\$s7 (salvos)
- Registradores floats: \$f2-\$f11 (temporários), \$f16-\$f32 (salvos)
- Stack: Para variáveis que não cabem em registradores
- Libertamos a memória ao sair de escopos
- Mapeamento completo de temporários para localizações na memória

CodeGen

Traduz IR para código MIPS:

- Operações aritméticas: soma, subtração, multiplicação, divisão (int/float)
- Bufferização na leitura de strings
- Uso da heap para strings dinâmicas
- Lazy evaluation para tudo o que envolva strings
- Função mips de conversão de ints, floats e booleans para string

- Função mips de concatenação de strings e de comparação (igualdade, buggy) e string builder (não foi completamente implementado, mas está a mais de meio do caminho)
- Função mips de exponenciação rápida (com verificação de overflow)

Bugs

Tinhamos muitas ideias, mas não tivemos o tempo necessário para fazer o debugging adequado. Preferimos experimentar de tudo, e assim continuaremos a fazer. Loops dificultaram a implementação de muitas das nossas grandes ideias, por exemplo. À custa de várias ideias, deixámos o debugging e optimizações bem feitas para o fim.

Como testar o código

Através da raíz do projeto, executar o script `./testValidation.sh`.

Correr `“./testValidation.sh -h”` para ajuda.

Para adicionar novos testes, deve adicionar na pasta `test_cases` os seguintes ficheiros:

- `[nome_teste].adb` (contém o código ada)
- `[nome_teste].expected` (contém o resultado esperado)
- `[nome_teste].input` (Opcional, contém o input que pretende escrever se houver chamadas `get_line`)

Referências

- Documentação oficial da linguagem Ada: <https://ada-lang.io>
- Wikipedia: [https://en.wikipedia.org/wiki/Ada_\(programming_language\)](https://en.wikipedia.org/wiki/Ada_(programming_language))
- Manual do Alex: <https://www.haskell.org/alex/>
- Manual do Happy: <https://www.haskell.org/happy/>
- Perplexity: <https://www.perplexity.ai/>
- ChatGPT: <https://chatgpt.com/>
- Material de apoio da disciplina de Compiladores (DCC-FCUP)