Universidade Federal de São Paulo
Disciplina: Compiladores
2° semestre de 2025
Grupo 6

Relatório Projeto Final

Aluno: Tales Miguel Machado Pereira
RA: 140247
Professor: Rodrigo Contreras

Dezembro
2025

Conteúdo

1 Objetivo

Implementar um compilador funcional para a linguagem didática C- (conforme o projeto do Apêndice A do livro de Kenneth C. Louden), contemplando:

- Scanner (analisador léxico);
- Tabela de Símbolos;
- Parser (analisador sintático) e construção da AST;
- Analisador semântico;
- Gerador de código intermediário em três endereços (sem otimizações).

A implementação deve ser em C ou C++. O uso de Flex (scanner) e Bison (parser) é recomendado, mas não obrigatório.

2 Manual de Uso

2.1 Requisitos

- GCC (C compiler)
- Flex (gerador de analisadores léxicos)
- Bison (gerador de analisadores sintáticos)
- Make

2.2 Instalação de Dependências

O projeto atual foi desenvolvido em ambiente Ubuntu 20.04/22.04 via WSL2. Portanto, utilizaremos comandos de instalação para Ubuntu/Debian:

```
sudo apt-get update
sudo apt-get install gcc flex bison make
```

2.3 Compilação

Para compilar o projeto:

```
make
```

Para limpar arquivos gerados:

```
make clean
```

Para recompilar do zero:

```
make clean
make
```

2.4 Execução

```
./cminus <arquivo.cm>

# Exemplo:
./cminus teste.cm
```

3 Arquitetura do Sistema

O compilador segue o pipeline clássico de compilação:

1. Código Fonte (.cm) - Arquivo de entrada na linguagem C-
2. Scanner (Flex) - Análise Léxica: reconhecimento de tokens
3. Parser (Bison) - Análise Sintática: construção da AST
4. Análise Semântica - Tabela de Símbolos e Verificação de Tipos
5. Gerador de Código - Produção de código intermediário de 3 endereços

3.1 Arquivos

```
- Makefile ------------------------ # Sistema de build
- README.md ----------------------- # Este arquivo
- globals.h ----------------------- # Definicoes globais
- cminus.l ------------------------ # Scanner (Flex)
- cminus.y ------------------------ # Parser (Bison)
- util.h / util.c ----------------- # Utilitarios AST
- symtab.h / symtab.c ------------- # Tabela de simbolos
- analyze.h / analyze.c ----------- # Analise semantica
- cgen.h / cgen.c ----------------- # Gerador de codigo
- main.c -------------------------- # Programa principal
- teste.cm ------------------------ # Arquivo de teste
```

3.1.1 Módulos Principais

- globals.h: Definições globais e estruturas de dados compartilhadas
- cminus.l: Scanner léxico (Flex)
- cminus.y: Parser sintático (Bison)
- util.c/util.h: Construção e impressão da AST
- symtab.c/symtab.h: Tabela de símbolos com pilha de escopos
- analyze.c/analyze.h: Análise semântica e verificação de tipos
- cgen.c/cgen.h: Gerador de código intermediário
- main.c: Programa principal

3.1.2 Pipeline de Execução

main.c executa o pipeline completo:

1. **Inicialização** (linhas 35-42): Abre arquivo fonte e configura `yyin` para o parser
2. **Parsing** (linha 47): Executa `yyparse()` gerando a AST
3. **Impressão da AST** (linha 62): Exibe estrutura hierárquica
4. **Análise Semântica** (linhas 66-73):

- `buildSymtab()`: constrói tabela de símbolos (linha 66)
- `typeCheck()`: verifica tipos (linha 73)

5. **Impressão da Tabela** (linhas 80-81): Exibe tabela completa e desempilha escopo global
6. **Geração de Código** (linha 83): Produz código intermediário

Em caso de erros em qualquer fase, a compilação é abortada (linhas 50-78) com mensagens descritivas.

4 Implementação

4.1 Scanner e Parser

As ferramentas **Flex** e **Bison** foram utilizadas neste projeto, dadas suas separações claras entre análise léxica (tokens) e sintática (gramática), sua facilidade de uso e o costume de uso adquirido no decorrer da disciplina.

4.2 Árvore Sintática Abstrata (AST)

A AST aqui implementada utiliza a estrutura **TreeNode** definida no arquivo *globals.h*, com:

- Ponteiros para filhos: Array de até 3 filhos (‘*MAXCHILDREN = 3*‘, linha 21), devido ao *if-then-else*:

```
if (condicao).....// child[0]
then_part.........// child[1]
else
else_part.........// child[2]
```

- Ponteiro sibling: Lista encadeada de irmãos para representar sequências (*struct treeNode* sibling*, linha 73)
- **Atributos flexíveis**: Unions para armazenar diferentes tipos de informação:
  - `kind`: tipo específico do nó (StmtKind, ExpKind, DeclKind) - linhas 76-80
  - `attr`: atributos do nó (operador, valor, nome) - linhas 83-87

– `type`: tipo de expressão (Integer, Void, IntegerArray, Boolean_) - linha 89

Esta estrutura permite representar a hierarquia do programa C- conforme especificado por Louden. A construção da AST é realizada pelas funções auxiliares em *util.c*:

- `newStmtNode()`: cria nós de comandos (linha 14)
- `newExpNode()`: cria nós de expressões (linha 32)
- `newDeclNode()`: cria nós de declarações (linha 50)

A impressão da árvore é feita pela função `printTree()` (*util.c* linha 82), que percorre recursivamente a estrutura exibindo a hierarquia com indentação de 4 espaços por nível.

4.3 Tabela de Símbolos

A tabela de símbolos é o componente responsável por armazenar e gerenciar informações sobre identificadores (variáveis, funções, parâmetros) ao longo do processo de compilação.

4.3.1 Arquitetura: Pilha de Escopos sobre Hash Table

Foi implementada uma **Pilha de Escopos (Scope Stack)** sobre uma estrutura de hash table. Esta arquitetura resolve problemas fundamentais de visibilidade e *shadowing* (ocultação de variáveis globais por variáveis locais de mesmo nome).

Componentes principais (definidos em *symtab.h* linhas 25-31 e *symtab.c* linha 12):

- **Hash Table**: Cada escopo mantém sua própria hash table de tamanho 211 (número primo escolhido para melhor distribuição, definido em *symtab.h* linha 11)
- **Estrutura ScopeListRec**: Representa um escopo individual com:

```
typedef struct ScopeListRec {
    char* scopeName; ............// Nome do escopo
    BucketList hashTable[SIZE]; // Hash table local
    struct ScopeListRec* parent; // Ponteiro p/ escopo pai
    struct ScopeListRec* next; ..// Lista de preservacao
    int nestedLevel; ............// Nivel de aninhamento
} *ScopeList;
```

- **scopeStack**: Pilha de escopos ativos (implementação com ponteiro `parent`) - *symtab.c* linha 11
- **allScopes**: Lista separada que preserva **todos** os escopos criados, mesmo após serem desempilhados, permitindo impressão completa da tabela de símbolos ao final - *symtab.c* linha 12

4.3.2 Operações Principais

Gerenciamento de Escopos:

- **st_push_scope()** (*symtab.c* linha 33): Cria um novo escopo filho do atual, incrementa nível de aninhamento, adiciona à lista `allScopes` e empilha em `scopeStack`
- **st_pop_scope()** (*symtab.c* linha 53): Desempilha o escopo atual, retornando ao escopo pai (mas mantém o escopo em `allScopes` para impressão posterior)

Operações de Busca:

- **st_lookup()** (*symtab.c* linha 116): Realiza **busca hierárquica** percorrendo do escopo atual até o global através dos ponteiros `parent`. Implementa a semântica de resolução de nomes: primeiro verifica escopo local, depois escopos externos progressivamente.
- **st_lookup_top()** (*symtab.c* linha 131): Busca **apenas no escopo atual**, utilizada para detectar redeclarações no mesmo escopo durante análise semântica.

Inserção e Impressão:

- **st_insert()** (*symtab.c* linha 61): Insere símbolo no escopo atual (topo da pilha) com tipo, linha de declaração e localização de memória
- **printSymTab()** (*symtab.c* linha 148): Percorre a lista `allScopes` (não `scopeStack`), imprimindo todos os escopos preservados com seus respectivos símbolos

4.3.3 Resolução do Problema de Shadowing

Esta arquitetura resolve o problema de *shadowing*, onde variáveis locais ocultam variáveis globais de mesmo nome. Considere o exemplo do arquivo *teste.cm*:

```
int x; /* Variavel GLOBAL */

void funcaoTeste(void) {
    int x; /* Variavel LOCAL */
    x = 55;
    output(x); /* Imprime 55 (x local) */
}

void main(void) {
    x = 10; /* Acessa x GLOBAL */
    output(x); /* Imprime 10 */
    funcaoTeste();
    output(x); /* Ainda imprime 10 (global nao alterado) */
}
```

A tabela de símbolos resultante mostra:

- **Escopo global**: `x` (tipo: int, MemLoc: 2, linha: 1)
- **Escopo funcaoTeste**: `x` (tipo: int, MemLoc: 0, linha: 5)
- **Escopo main**: `i` (tipo: int, MemLoc: 0, linha: 13)

Quando `st_lookup("x")` é chamado dentro de `funcaoTeste`, a busca hierárquica encontra primeiro o `x` local (MemLoc 0), ocultando corretamente o `x` global (MemLoc 2). Fora de `funcaoTeste`, a busca encontra apenas o `x` global.

4.4 Análise Semântica

A análise semântica verifica a correção do programa além da sintaxe, validando tipos, declarações e uso de identificadores. Implementada no arquivo *analyze.c*, é dividida em dois passes principais:

4.4.1 Primeiro Pass: buildSymtab

A função `buildSymtab()` (*analyze.c* linha 144) percorre a AST e popula a tabela de símbolos. O processo inclui:

- **Inicialização**: Cria escopo global e insere funções built-in `input` e `output` (linhas 145-147)
- **Travessia**: Utiliza a função `traverse()` (linha 19) que percorre a AST executando:
  - `insertNode()`: processa cada nó antes dos filhos (linha 44)
  - `afterInsertNode()`: processa após os filhos (linha 130)
- **Gerenciamento de escopos**:
  - Push de escopo ao entrar em função (`FunK`, linha 72)
  - Pop de escopo ao sair de função (`afterInsertNode`, linha 135)
- **Validações realizadas**:
  - Redeclaração de variáveis no mesmo escopo (linhas 51-57)
  - Variáveis declaradas com tipo `void` (linhas 58-62)
  - Redeclaração de funções (linhas 68-73)
  - Uso de identificadores não declarados (linhas 105-115)

4.4.2 Segundo Pass: typeCheck

A função `typeCheck()` (*analyze.c* linha 295) realiza verificação de tipos em dois sub-passes:

Sub-pass 1: setNodeTypes (linhas 151-179) Propaga tipos pela AST atribuindo tipos aos nós de expressão:

- Constantes (`ConstK`): tipo `Integer` (linha 157)
- Identificadores (`IdK`): busca tipo na tabela de símbolos (linha 161)
- Arrays indexados: tipo `Integer` mesmo que variável seja `IntegerArray` (linha 165)
- Operações (`OpK`): tipo `Integer` (linha 169)
- Chamadas (`CallK`): tipo da função retornada (linha 173)

Sub-pass 2: checkNode (linhas 191-288) Valida compatibilidade de tipos:

- **Atribuições** (`AssignK`, linha 195):
  - Arrays sem índice em atribuição (linha 197)
  - Atribuição de expressões `void` (linha 204)
- **Operações** (`OpK`, linha 217):
  - Operandos não podem ter tipo `void` (linhas 218-227)
- **Identificadores** (`IdK`, linha 233):
  - Indexação de variáveis não-array (linha 237)

4.4.3 Validação Final

Após os dois passes, verifica-se a existência da função `main` (linhas 300-303). Caso não declarada, o programa emite erro semântico e encerra.

4.5 Geração de Código Intermediário

O gerador de código produz código intermediário de três endereços sem otimizações, conforme especificação do Louden. Implementado no arquivo *cgen.c*, gera código linear para posteriormente ser gerado algum código de máquina.

4.5.1 Elementos Básicos

- **Temporários**: Gerados sequencialmente (`t0, t1, t2...`) pela função `newTemp()` (linha 17)
- **Labels**: Numerados sequencialmente (`L0, L1, L2...`) pela função `newLabel()` (linha 27) para controle de fluxo

4.5.2 Funções Principais

cGenExp() (linha 36) Gera código para expressões, retornando o nome do temporário contendo o resultado:

- **Constantes** (`ConstK`, linha 47): Atribui valor à temporária (`t0 = 5`)
- **Identificadores** (`IdK`, linha 51):

  - Variáveis simples: retorna o nome da variável
  - Arrays indexados: gera acesso `t0 = arr[index]`

- **Operadores** (`OpK`, linha 58): Gera código de três endereços (`t0 = t1 + t2`)
- **Chamadas de função** (`CallK`, linha 105):
  - Gera *param* para cada argumento
  - Gera `t0 = call func, N` onde N é o número de argumentos

cGenStmt() (linha 121) Gera código para comandos:

- **Atribuição** (`AssignK`, linha 133): `var = expr` ou `arr[index] = expr`
- **If/If-Else** (`IfK`, linha 144):

```
if_false condicao goto L0
    codigo_then
    goto L1
L0:
    codigo_else
L1:
```

- **While** (`WhileK`, linha 158):

```
L0:
    if_false condicao goto L1
    corpo_loop
    goto L0
L1:
```

- **Return** (`ReturnK`, linha 168): `return expr` ou `return`
- **Compound** (`CompoundK`, linha 176): Processa sequência de statements usando **while loop** (linhas 179-188) sobre siblings

cGenDecl() (linha 204) Gera código para declarações:

- **Funções** (`FunK`, linha 208): Formato:

```
func nome:
    param p1
    param p2
    corpo
endfunc
```

- **Arrays** (`ArrayK`, linha 222): `array nome[tamanho]`

cGenExpStmt() (linha 234) Trata expressões usadas como statements (chamadas de função sem atribuição). Gera formato `call func, N` com contagem de argumentos (linha 245).

4.5.3 Detalhe de Implementação

Em `CompoundK`, o processamento de siblings utiliza **while loop** (linhas 179-188) e **não** recursão. Esta decisão evita duplicação de código que ocorreria ao combinar loop e recursão sobre a mesma estrutura de siblings.

4.5.4 Formato de Chamadas

Chamadas de função seguem o formato `call funcao, N`, onde N indica explicitamente o número de argumentos. Exemplo:

```
param a
call output, 1
```

Este formato facilita a posterior geração de código de máquina que necessita configurar a pilha corretamente.

4.6 Geração de Código Intermediário

O gerador de código produz código intermediário de três endereços sem otimizações, conforme especificação do Louden. Implementado no arquivo *cgen.c*, gera código linear adequado para posterior geração de código de máquina.

4.6.1 Elementos Básicos

- **Temporários**: Gerados sequencialmente (`t0, t1, t2...`) pela função `newTemp()` (linha 17)

- **Labels**: Numerados sequencialmente (`L0, L1, L2...`) pela função `newLabel()` (linha 27) para controle de fluxo

4.6.2 Funções Principais

cGenExp() (linha 36) Gera código para expressões, retornando o nome do temporário contendo o resultado:

- **Constantes** (`ConstK`, linha 47): Atribui valor à temporária (`t0 = 5`)
- **Identificadores** (`IdK`, linha 51):
  - Variáveis simples: retorna o nome da variável
  - Arrays indexados: gera acesso `t0 = arr[index]`
- **Operadores** (`OpK`, linha 58): Gera código de três endereços (`t0 = t1 + t2`)
- **Chamadas de função** (`CallK`, linha 105):
  - Gera `param` para cada argumento
  - Gera `t0 = call func, N` onde N é o número de argumentos

cGenStmt() (linha 121) Gera código para comandos:

- **Atribuição** (`AssignK`, linha 133): `var = expr` ou `arr[index] = expr`
- **If/If-Else** (`IfK`, linha 144):

```
if_false condicao goto L0
    codigo_then
    goto L1
L0:
    codigo_else
L1:
```

- **While** (`WhileK`, linha 158):

```
L0:
    if_false condicao goto L1
    corpo_loop
    goto L0
L1:
```

- **Return** (`ReturnK`, linha 168): `return expr` ou `return`
- **Compound** (`CompoundK`, linha 176): Processa sequência de statements usando **while loop** (linhas 179-188) sobre siblings

cGenDecl() (linha 204) Gera código para declarações:

- **Funções** (`FunK`, linha 208): Formato:

```
func nome:
    param p1
    param p2
    corpo
endfunc
```

- **Arrays** (`ArrayK`, linha 222): `array nome[tamanho]`

cGenExpStmt() (linha 234) Trata expressões usadas como statements (chamadas de função sem atribuição). Gera formato `call func, N` com contagem de argumentos (linha 245).

Em `CompoundK`, o processamento de siblings utiliza **while loop** (linhas 179-188) e **não** recursão. Anteriormente foi escolhida uma abordagem via recursão, mas que acarretou na duplicação de código de saída, pois combinava loop e recursão sobre uma mesma estrutura de siblings. Utilizando o **while**, isto não ocorre.

4.6.3 Formato de Chamadas

Chamadas de função seguem o formato `call funcao, N`, onde N indica explicitamente o número de argumentos. Exemplo:

```
param a
call output, 1
```

Este formato facilita a posterior geração de código de máquina que necessita configurar a pilha corretamente.

4.7 Saída do Compilador

O compilador produz três saídas principais, controladas pelo programa principal em *main.c* (linhas 26-91):

1. **Árvore Sintática Abstrata** (linha 62): Representação hierárquica do programa gerada pela função `printTree()` de *util.c*. Exibe a estrutura completa do código fonte com indentação mostrando níveis de aninhamento.

2. **Tabela de Símbolos** (linha 80): Símbolos organizados por escopo (global, funções) com informações de tipo, localização de memória (MemLoc) e linhas de uso. Gerada pela função `printSymTab()` após análise semântica completa.

3. **Código Intermediário** (linha 83): Código de três endereços sem otimizações, pronto para posterior geração de código de máquina. Produzido pela função `codeGen()`.

Exemplos completos das três saídas são apresentados na Seção 5.

5 Testes e Resultados

Esta seção apresenta a execução do compilador sobre o arquivo de teste *teste.cm*, demonstrando o funcionamento completo de todas as fases da compilação.

5.1 Arquivo de Entrada

O arquivo *teste.cm* foi projetado para testar o tratamento de *shadowing* (variável local ocultando global de mesmo nome):

```
int x; /* Variavel GLOBAL */
int y;

void funcaoTeste(void) {
    int x; /* Variavel LOCAL (oculta a global) */

    x = 55;
    output(x); /* Deve imprimir 55 */
}

void main(void) {
    int i;

    x = 10; /* Acessando o x GLOBAL */
    y = 20;

    output(x); /* Deve imprimir 10 */

    funcaoTeste(); /* x local = 55 */

    output(x); /* x global continua 10 */
}
```

Objetivo do teste: Verificar se o compilador mantém corretamente dois símbolos `x` distintos (um global, um local em `funcaoTeste`), sem que a atribuição ao `x` local afete o `x` global.

5.2 Execução

Comando de execução:

```
./cminus teste.cm
```

5.3 Árvore Sintática Abstrata (Trecho)

A AST gerada mostra a estrutura hierárquica do programa:

```
Declaracao de Variavel: x tipo int
  Tipo: int

Declaracao de Variavel: y tipo int
  Tipo: int

Declaracao de Funcao: funcaoTeste tipo void
  Parametros: void
  Corpo (Compound):
    Declaracao de Variavel: x tipo int (LOCAL)
      Tipo: int
    Atribuicao:
      Id: x
      Const: 55
    Expressao (Call):
      Id: output
        Id: x

Declaracao de Funcao: main tipo void
  Parametros: void
  Corpo (Compound):
    Declaracao de Variavel: i tipo int
      Tipo: int
    Atribuicao:
      Id: x (GLOBAL)
      Const: 10
    Atribuicao:
      Id: y
      Const: 20
    Expressao (Call):
      Id: output
        Id: x
    Expressao (Call):
      Id: funcaoTeste
```

```
  Expressao (Call):
    Id: output
      Id: x
```

5.4 Tabela de Símbolos

A tabela de símbolos demonstra o correto gerenciamento de escopos e *shadowing*:

```
Escopo: main (nivel 1)
Nome Tipo MemLoc Linhas
****************************************************
i int 0 12

Escopo: funcaoTeste (nivel 1)
Nome Tipo MemLoc Linhas
****************************************************
x int 0 5 7 8

Escopo: global (nivel 0)
Nome Tipo MemLoc Linhas
****************************************************
funcaoTeste void 3 4 19
main void 4 11
y int 2 2 15
x int 1 1 14 17 21
output void 1 -1
input int 0 -1
```

5.4.1 Análise da Tabela

1. **Shadowing correto**: Existem dois símbolos `x` distintos:
   - `x` global (MemLoc 1): declarado linha 1, usado nas linhas 14, 17, 21
   - `x` local em `funcaoTeste` (MemLoc 0): declarado linha 5, usado nas linhas 7, 8
2. **Escopos preservados**: Todos os três escopos (global, funcaoTeste, main) são exibidos corretamente na tabela, mesmo após serem desempilhados

3. **Built-ins**: Funções `input` e `output` inseridas automaticamente no escopo global (linha -1)
4. **Níveis de aninhamento**: Escopo global (nível 0), funções (nível 1)
5. **Localização de memória**: Cada escopo mantém seu próprio contador de MemLoc

5.5 Código Intermediário de Três Endereços

O código gerado segue o formato linear sem otimizações:

```
func funcaoTeste:
x = 55
param x
call output, 1
endfunc

func main:
x = 10
y = 20
param x
call output, 1
call funcaoTeste, 0
param x
call output, 1
endfunc
```

5.5.1 Análise do Código Gerado

Aspectos avaliados:

1. **Formato**: Cada função delimitada por `func/endfunc`
2. **Chamadas com contagem**: Todas as chamadas incluem número de argumentos:
   - `call output, 1`: função output recebe 1 parâmetro
   - `call funcaoTeste, 0`: função sem parâmetros
3. **Atribuições diretas**: `x = 55`, `y = 20` (sem temporários desnecessários)
4. **Sem duplicações**: Código gerado uma única vez por statement
5. **Ordem**: Statements gerados na ordem de execução do programa

5.6 Validação Final

O teste comprova que o compilador implementa corretamente todas as fases: análise léxica, análise sintática, construção da AST, tabela de símbolos com pilha de escopos, análise semântica e geração de código intermediário.

6 Conclusão

Os testes realizados validam o funcionamento correto de todas as fases: análise léxica (scanner), análise sintática (parser), construção da AST, análise semântica com tabela de símbolos hierárquica e geração de código. O tratamento de shadowing demonstra o correto gerenciamento de escopos através da pilha implementada.

A implementação permitiu observar na prática os conceitos estudados na disciplina de Compiladores, desde o reconhecimento de tokens até a geração de código intermediário, evidenciando a integração entre as diferentes fases do processo de compilação.

Bibliografia

LOUDEN, K. C. Compiler Construction: Principles and Practice. Boston, EUA, PWS Publishing Company. 1997.