



Universidade Federal de Uberlândia  
Faculdade de Computação



# Projeto da Disciplina

Curso de Bacharelado em Ciência da Computação  
GBC071 - Construção de Compiladores  
Prof. Luiz Gustavo Almeida Martins

# Visão Geral do Projeto

- Foco no **front-end** do compilador
  - Não será implementada as etapas de otimização e síntese
  - Será adotado algum *back-end* disponível no ambiente de compilação
- Uso do ambiente de compilação **LLVM**
  - Deve-se gerar uma IR compatível com o ambiente
- Características gerais:
  - **Analizador léxico será uma subrotina** chamada pelo analisador sintático
    - Deve retornar o próximo *token* da cadeia de entrada (código-fonte) a cada chamada
  - **Compilador de um passo:**
    - Todas as demais fases do *front-end* serão implementadas em um único módulo
    - Tradução dirigida por sintaxe
  - **Semântica estática:**
    - Realizada em tempo de compilação



Universidade Federal de Uberlândia  
Faculdade de Computação



# **Especificação da Linguagem**

# Componentes Básicos

- Estrutura principal:

**Sintaxe:**    *procedure principal*  
                  *bloco*

– *procedure principal* funciona como o *int main()* do C

- Bloco:

**Sintaxe:**        *Inicio*  
                      *declaração das variáveis*  
                      *sequência de comandos*  
                      *Fim*

# Componentes Básicos

- **Declaração de variáveis:**

**Sintaxe:** *tipo : <lista\_ids>;*

- **Tipo** define o tipo de dado da variável
  - Usaremos os tipos: ***integer***, ***char*** e ***real***
- **<lista\_ids>** é formada por um ou mais identificadores de variáveis separados por vírgula
  - **Ex:** *integer: x; real : y, z; char : w, c, a;*

- **Comando de seleção**

**Sintaxe:** ***se (cond) entao***

*bloco*

***senao***

*bloco*

# Componentes Básicos

- **Comentários:**

**Sintaxe:**    *{ texto do comentário }*

- **Comandos de repetição:**

**Sintaxe (*while*):**        *enquanto (condição)*  
                                  *bloco*

**Sintaxe (*do-while*):**    *repita*  
                                  *bloco*  
                                  *enquanto (condição);*

- **Comando de atribuição:**

**Sintaxe:**    *id := expressao;*

# Componentes Básicos

- **Condições:**

- Permite **operadores relacionais**

- Igual (==), diferente (<>), menor (<), maior (>), menor ou igual (<=), maior ou igual (>=)

- **Expressões:**

- Permite **operadores aritméticos**

- Soma (+), subtração (-), multiplicação (\*), divisão (/)

- Permite **constantes** compatíveis com os tipos definidos na linguagem (*integer*, *char*, *real*)

- ***char*** deve estar entre apóstrofo (ex: 'A')
    - ***integer*** deve estar entre -32768 e +32767
    - ***real*** pode ser ponto fixo (ex: 5.3) ou notação científica (ex: 0.1E-2)

- Permite parênteses para priorizar operações



Universidade Federal de Uberlândia  
Faculdade de Computação



# **Etapas do Projeto**



# 1a Etapa do Projeto

- **Especificação da linguagem:**
  - Definição da **gramática livre de contexto (GLC)** com as estruturas da linguagem especificada
  - Identificação dos ***tokens*** usados na gramática
    - Apresentar uma tabela com o nome e o tipo do atributo que será retornado para cada token
  - Definição dos padrões (**expressões regulares**) para os lexemas aceitos em cada *token*
- Gerar um arquivo pdf do relatório com a Seção **“Projeto da Linguagem”** contendo as informações acima

# 2a Etapa do Projeto

- **Análise Léxica:**
  - Especificação dos **diagramas de transição** para cada *token*
    - Conversão expressões regulares em autômatos finitos não determinístico (AFND)
    - Unificação em um único AFND
    - Conversão do AFND único em um autômato finito determinístico (AFD)
    - Conversão do AFD no diagrama de transição do analisador léxico
  - Implementação manual do **analisador léxico**
    - Subrotina chamada pelo analisador sintático (devolve um *token* por chamada)
    - Deve usar a abordagem **dirigida por tabela**
    - Deve retornar para cada *token*: nome (tipo), valor do atributo (quando necessário) e a posição (linha e coluna do início do lexema encontrado)
    - Tabela de símbolos armazena identificadores e constantes numéricas
- Incluir ao relatório a Seção “**Análise Léxica**” com os resultados da etapa de construção do diagrama de transição
- Gerar um arquivo compactado com o relatório resultante e os códigos do analisador léxico

# 3a Etapa do Projeto

- **Análise Sintática:**

- Determinar se a gramática da linguagem é do tipo **LL(1)**. Se não for, fazer os ajustes necessários:
  - Remoção de recursão a esquerda
  - Tratamento de ambiguidades (ex: fatoração)
- Calcular **FIRST** e **FOLLOW** para os símbolos da gramática
- Construção dos **grafos sintáticos**
- Implementação manual de um **analisador sintático preditivo**
  - Deve usar a abordagem baseada em **descida recursiva**
  - Deve gerar a árvore sintática correspondente

- Incluir ao relatório a Seção “**Análise Sintática**” com os resultados das 3 primeiras etapas
- Gerar um arquivo compactado com o relatório resultante e os códigos do analisador sintático

# 4a Etapa do Projeto

- **Tradução dirigida por sintaxe:**
  - **Análise semântica**
    - Verificação de tipos (compatibilidade e declaração prévia)
    - Se necessário, realiza coerção automática ( $\text{int} \rightarrow \text{real}$ )
  - **Geração do código intermediário**
    - Reproduzir a representação intermediária do LLVM
  - **Etapas:**
    - Definição dos atributos
    - Especificação dos esquemas de tradução
    - Incorporar no analisador sintático as ações semânticas
- Incluir ao relatório a Seção “**Tradução dirigida por sintaxe**” com os resultados das 2 primeiras etapas
- Gerar uma arquivo compactado com o relatório resultante e os códigos do analisador sintático modificado

# Definição do Código Intermediário

- Para cada elemento estrutural da linguagem, verificar como é a IR correspondente no LLVM
  - **1º passo:** construir um **programa vazio** em C (sem declarações e comandos na *main()*) e executar o **front-end CLANG** e analisar a IR gerada
  - **2º passo:** incluir cada estrutura pretendida e verificar as mudanças na IR
- Incluir ao relatório a Seção “**IR do LLVM**” com as IR obtidas a partir de cada instrução da linguagem

# Definição do Código Intermediário (Exemplo)



```
int main()
```

```
{  
  
}
```

```
/* Código fonte */
```

```
; ModuleID = 'teste.c'
```

```
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
```

```
target triple = "x86_64-unknown-linux-gnu"
```

```
; Function Attrs: nounwind uwtable
```

```
define i32 @main() #0 {
```

```
    ret i32 0
```

```
}
```

```
attributes #0 = { nounwind uwtable "disable-tail-calls"="false" "less-  
precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-  
elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "  
"stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-  
features"="+sse,+sse2" "unsafe-fp-math"="false" "use-soft-  
float"="false" }
```

```
!llvm.ident = !{!0}
```

```
!0 = !{"clang version 3.7.1 (tags/RELEASE_371/final)"}
```

```
/* Código intermediário (SSA) */
```

# Definição do Código Intermediário (Exemplo)

```
int main()
```

```
{
```

```
    int x;
```

```
}
```

```
/* Código fonte */
```

```
; ModuleID = 'teste.c'
```

```
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
```

```
target triple = "x86_64-unknown-linux-gnu"
```

```
; Function Attrs: nounwind uwtable
```

```
define i32 @main() #0 {
```

```
    %x = alloca i32, align 4
```

```
    ret i32 0
```

```
}
```

```
attributes #0 = { nounwind uwtable "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+sse,+sse2" "unsafe-fp-math"="false" "use-soft-float"="false" }
```

```
!llvm.ident = !{!0}
```

```
!0 = !{"clang version 3.7.1 (tags/RELEASE_371/final)"}
```

```
/* Código intermediário (SSA) */
```



Universidade Federal de Uberlândia  
Faculdade de Computação



# **Ambiente de Compilação**



# Ambiente de Compilação

- **LLVM** (site: <https://llvm.org/>)
  - Execução dos componentes (*toolchains*) por linha de comando
    - Similar ao GCC
  - Usa *flags* para direcionar/personalizar a compilação
    - Ex: *-lm* para funções matemáticas
  - Plataformas suportadas (fonte: *llvm.org*):

| OS             | Arquitetura     | Compiladores  |
|----------------|-----------------|---------------|
| Linux          | x861            | GCC, Clang    |
| Linux          | amd64           | GCC, Clang    |
| Linux          | ARM             | GCC, Clang    |
| Linux          | PowerPC         | GCC, Clang    |
| Solaris        | V9 (Ultrasparc) | GCC           |
| FreeBSD        | x861            | GCC, Clang    |
| FreeBSD        | amd64           | GCC, Clang    |
| NetBSD         | x861            | GCC, Clang    |
| NetBSD         | amd64           | GCC, Clang    |
| MacOS2         | PowerPC         | GCC           |
| MacOS          | x86             | GCC, Clang    |
| Win32 (Cigwin) | x861, 3         | GCC           |
| Windows        | x861            | Visual Studio |
| Win64          | x86-64          | Visual Studio |

# Ambiente de Compilação

- **Compilação direta:**

- Sintaxe: **clang -o exeCode sourceCode.c**

- **Compilação em etapas:**

- **Análise (front-end):**

- Sintaxe: **clang sourceCode.c -emit-llvm -S -o IRCode.ll**
- **-emit-llvm** deve ser usado com as opções **-S** para gerar IR (.ll) ou **-c** para gerar *bitcode* (.bc)

- **Otimização (middle-end):**

- Sintaxe: **opt <seq> IRCode.ll -S -o IRCodeOptim.ll**
- **<seq>** representa a sequência de otimização que deve ser aplicada na IR
  - Ex: -O1, -O2, -O3, “-tti -tbaa -verify -domtree -sroa -early-cse -basicaa -aa -gvn-hoist”

- **Síntase (back-end):**

- Código Assembly: **llc IRCode.ll -o asmCode.s**
- Código de máquina: **clang -o exeCode IRCode.ll** OU  
**clang -o exeCode asmCode.s** OU  
**gcc asmCode.s -o exeCode** (alternativa com GCC)