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

Características Gerais do Projeto

Foco no *front-end* do compilador

−Análise léxica, sintática, semântica e geração do código intermediário (IR)

Analisador léxico é uma subrotina do analisador sintático

−Retorna um ÚNICO token do código-fonte a cada chamada (próximo token)

Tradução dirigida por sintaxe:

−Demais etapas do front-end (análise semântica e geração da IR) são executadas durante a análise sintática (**único módulo**)

Analisadores devem ser implementados manualmente:

−Não deve usar geradores automáticos, nem inteligência artificial

Universidade Federal de Uberlândia
Faculdade de Computação

Especificação da Linguagem

Estrutura Principal

- **Sintaxe:**

tipo main ( )

Bloco

- **tipo:** define o tipo de retorno da função
  - Usaremos os tipos: **void**, **int**, **char** e **float**

- **main:** palavra reservada que identifica a função principal do programa

- ***Bloco*:** estrutura formada por declaração das variáveis (opcional) e pela sequência de comandos (nessa ordem)

  Sintaxe:

  [

  declaração das variáveis

  sequência de comandos

  ]

Declaração de Variáveis

- **Sintaxe:**

  tipo *lista_ids* **;**

  - **tipo:** define o tipo de dado da variável
    - Usaremos os tipos: **void**, **int**, **char** e **float**

  - ***lista_ids***: especifica 1 ou + nomes (id) de variáveis separados por vírgula
    - **Ex:** int idade;

      float nota, z ;

      char c, letra, s;

Comando de Atribuição

- **Sintaxe:**

  id := *expressão* **;**

- **Expressão:**
  - Permite **operadores aritméticos**
    - Soma (**+**), subtração (**-**), multiplicação (*), divisão (*/*) e potência (**)
  - Permite **constantes** compatíveis com os tipos definidos:
    - **char** deve estar entre apóstrofo (ex: ‘A’)
    - **int** deve estar entre 0 e 32767
    - **float** pode ser ponto fixo (ex: 5.3) ou notação científica (ex: 0.1E-2)
    - Negação de constantes numéricas (int e float) deve ser tratada como operador unário
  - Permite **parênteses** para priorizar operações

Comando de Seleção

- **Sintaxe:**

if (*condição*) then

comando ou bloco

elsif (*condição*) then

comando ou bloco

else

comando ou bloco

AMBAS AS PARTES SÃO OPCIONAIS

(pode ter ou não)

Obs: *comando* permite uma única instrução

Comandos de Repetição

- **Comando while (sintaxe):**

  while (*condição***) do**

  comando ou bloco

- **Comando do (sintaxe):**

  do

  comando ou bloco

  while (*condição***) ;**

- **Condição:**
  - Permite apenas **operadores relacionais**
    - Igual (**==**), diferente (**!=**), menor (**<**), maior (**>**), menor ou igual (**<=**), maior ou igual (**>=**)

Comandos de Repetição

- **Comando for (sintaxe):**

for (id ; num ; num ; *expressão* **)**

comando ou bloco

- **id:** especifica o nome (id) da variável de controle
- **num:** especifica as constantes inteiras que determinam os valores inicial (primeira ocorrência) e final (segunda ocorrência) da variável de controle
- ***expressao*:** define a expressão aritmética que modifica a variável de controle a fim de atingir o valor final a partir do inicial

Comentários

- **Sintaxe:**

{% *texto_comentario* **%}**

- **Observações:**
  - Embora tenha estrutura, destaca-se que comentários são elementos léxicos (*token*)
    - Não pode ser dividido em componentes menores
    - Não deve ser especificado, nem tratado na análise sintática
  - Pode ocupar mais de uma linha do código fonte
    - Similar ao comentário (/**/) da linguagem C

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 de atributo que será retornado (quando aplicável) para cada *token*
  - Definição dos padrões (**expressões regulares**) de cada *token* (inclusive os *tokens* especiais)
- Gerar um relatório (arquivo pdf) com a Seção "**Projeto da Linguagem**", contendo as informações acima

2a Etapa do Projeto

- **Análise Léxica (especificação):**
  - Elaboração do **diagrama de transição**
    - Gerar um diagrama de transição para cada *token*
    - Unificá-los em um diagrama não determinístico
    - Convertê-lo em um diagrama de transição determinístico

- Incluir a Seção "**Análise Léxica**" no relatório da etapa anterior, apresentando os artefatos gerados durante o processo de construção do diagrama de transição

2a Etapa do Projeto

- **Análise Léxica (implementação):**
  - Subrotina chamada pelo analisador sintático
    - Devolve um único *token* por vez
    - Deve retornar o tipo do *token*, valor do atributo (quando necessário) e a posição (linha e coluna do início do lexema)
  - Implementação **codificação direta**
  - Preencha **tabela de símbolos** com identificadores e constantes
    - **Campos**: nome do token, lexema e tipo do dado
      - O último campo só será preenchido quando aplicável (constantes)
  - Tratamentos especial para **comentários e separadores**
  - Emite mensagem de erro "útil" quando identificar **erros léxicos**

- Compactar o relatório e os códigos do analisador léxico no arquivo ***AnaliseLexica_NomeAluno.zip***

3a Etapa do Projeto

- **Análise Sintática (especificação):**
  - Fazer os ajustes necessários para que a GLC da linguagem seja **LL(1)**:
    - Tratamento de ambiguidades (associatividade e precedência)
    - Remoção de recursão a esquerda
    - Tratamento do não determinismo (fatoração à esquerda)

    ex: associatividade e precedência, fatoração, etc.
  - Calcular **FIRST** e **FOLLOW** para os símbolos da gramática
  - Construção da **tabela de análise preditiva**
- Incluir a Seção "**Análise Sintática**" no relatório da etapa anterior, com os resultados os processos acima

3a Etapa do Projeto

- **Análise Sintática (implementação):**
  - Implementação manual de um **analisador sintático preditivo**:
    - Utilizar a abordagem **baseada em tabela preditiva**
  - Construir a árvore sintática concreta (**árvore de derivação**):
    - Utilizar uma **estrutura encadeada** para a árvore
  - Emitir mensagem de erro “útil” quando for identificado um **erro sintático**
- Compactar o relatório e os códigos do analisador sintático no arquivo ***AnaliseSintatica_NomeAluno.zip***

4a Etapa do Projeto

- **Tradução dirigida por sintaxe:**
  - **Análise semântica**
    - Verificação de tipos (declaração prévia e compatibilidade de operandos)
    - Se necessário, realiza a coerção automática (ex: int → float)
    - Complementar a **tabela de símbolos** com o tipo dos identificadores
  - **Geração do código intermediário**
    - Construir a representação intermediária do LLVM
  - **Etapas:**
    - Definição dos atributos para a GLC
    - Especificação dos esquemas de tradução
    - Incorporar as ações semânticas no código do analisador sintático
- Incluir ao relatório a Seção "**Tradução dirigida por sintaxe**" com os resultados das 2 primeiras etapas
- Compactar o relatório e os códigos do *front-end* no arquivo ***FrontEnd_NomeAluno.zip***

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 instrução/estrutura pretendida e verificar as mudanças na IR para mapear como deve ser a tradução

Definição do Código Intermediário (Exemplo)


```
; 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 (alterativa com GCC)