Análise de Algoritmos - Ordenação

Gustavo de Souza Silva
Guilherme de Souza Silva
Arthur Xavier
Shumaiquer Souto

Faculdade de Computação
Universidade Federal de Uberlândia

20 de junho de 2017

Lista de Listagens

Sumário

Capítulo 1

Introdução

Este documento foi escrito como relatório de uma tarefa com objetivo de aumentar minhas experiências de uso da linguagem fonte MiniC, possibilitar uma compreensão mais aprofundada dos códigos Assembly da plataforma LLVM (Low Level Virtual Machine) e desenvolver habilidade de uso das ferrramentas para a plataforma LLVM.

1.1 LLVM

Desenvolvida para otimizar em tempo de compilação, ligação(linker) e execução de programas escritos em várias linguagems de programação, LLVM (Low Level Virtual Machine) é uma infraescrutura de compilador escrita em C++. Inicialmente foi feita para C e C++, porém foi expandida para Java, Python, Ruby, Haskell e muitas outra linguagens.

1.2 LLVM IR

LLVM IR (LLVM Intermediate Representation) É o aspecto mais importante de um projeto, ele é a forma como é representado o código no compilador. Ele é o código intermediário do compilador, foi projetado para analisar e otimizar o código que foi compilado.

1.3 Instalação

Alguns pacotes devem ser instalados no linux Ubuntu para que seja feita a compilação e geração do código Assembly da LLVM.

```
> sudo apt-get install llvm
> sudo apt-get install clang
```

Com o primeiro comando a llvm é instalada no sistema operacional, no segundo comando é instalada a interface clang, que faz o front-end do C para LLVM

Agora, podemos também instalar o OCAML, linguagem que será utilizada para fazermos o compilador, para isto digite:

```
> sudo apt-get install ocaml
> sudo apt-get install rlwrap
```

O primeiro comando instala o ocaml no computador, já o segundo instala o rlwrap, que salva comandos executados no terminal e se faz muito útil neste documento.

1.4 Compilando C em LLVM-IR e executado

Para que seja feita a conversão de um código em linguagem C para LLVM-IR, deve ser feito o seguinte comando no terminal:

```
> clang -S -emit-llvm nome.c -o nome.ll
```

Onde nome.c é o nome do arquivo c e nome.ll é o nome do arquivo de saída no código Assembly da LLVM.

Para que seja feita a execução na LLVM, os seguintes comandos devem ser seguidos:

```
> llvm-as nome.ll
> lli nome.bc
```

Onde o primeiro comando cria o arquivo binário, e o segundo o executa na llvm.

Para criar o arquivo obj, utilizando o arquivo .bc do llvm, o seguinte comando é necessário:

```
> llc -filetype=obj nome.bc
```

Após a criação do arquivo obj com o llc, o arquivo executável pode ser criado utilizando o GCC.

```
> gcc nome.o
> ./a.out
```

Como nota de observação, temos que podemos gerar um arquivo executavel em C diretamente pelo GCC utilizando os comandos:

```
> gcc nome.c -o nome.o
> ./nome
```

O primeiro comanda cria o arquivo .o (executavel) e o segundo faz sua execução.

Capítulo 2

Programas de Teste

A seguir, os programas em C, seguidos por sua tradução para Assembly da LLVM, no fim deste capítulo, é póssivel encontrar uma "tradução"para as intruções do assembly:

2.1 Nano Programas

2.2 Micro Programas

2.3 Instruções Importantes

2.3.1 Tipos do LLVM

Primitivos:
i1,i2,...,i8,....,i16,...,i32
label,void
float e double

Derivados
Array: [40 x i32]
Pointers: [4 x i8]*
Structures: i32,(i32)*,i1
Function - <tiporetorno>(<parametro>):i32(i32)

2.3.2 Condicionais

Operadores possíveis condicionais de icmp
eq: Igual
ne: Diferente
ugt: unsigned Maior que

uge: unsigned Maior ou igual
ult: unsigned Menor que
ule: unsigned Menor ou igual
sgt: signed Maior que
sge: signed Maior ou igual
slt: signed Menor ou igual
sle: signed less or equal

Operadores possíveis condicionais de fcmp:
false: Sempre retorna falso, não importanto a operação.
true: Sempre retorna true, não importando a operação.
oeq: retorna true se todos os operadores não são QNAN e op1 é igual a op2.
ogt: retorna true se todos os operadores não são QNAN e op1 é maior que op2.
oge: retorna true se todos os operadores não são QNAN e op1 é maior ou igual que op2.
olt: retorna true se todos os operadores não são QNAN e op1 é menor que op2.
ole: retorna true se todos os operadores não são QNAN e op1 é menor ou igual que op2.
one: retorna true se todos os operadores não são QNAN e op1 é diferente de op2.
ord: retorna true se todos os operadores não são QNAN.
ueq: retrona true se qualquer operador é QNAN ou op1 é igual a op2.
ugt: retrona true se qualquer operador é QNAN ou op1 é maior que op2.
uge: retrona true se qualquer operador é QNAN ou op1 é maior ou igual que op2.
ult: retrona true se qualquer operador é QNAN ou op1 é menor que op2.
ule: retrona true se qualquer operador é QNAN ou op1 op1 é menor ou igual que op2.
une: retrona true se qualquer operador é QNAN ou op1 é diferente de op2.
uno: retrona true se qualquer operador é QNAN.

2.4 Programas Extras

Alguns programas extras para complementar a leitura.

Capítulo 3

Compilador

O Compilador é um programa que traduz um programa de entrada de uma linguagem fonte para uma linguagem que possa ser indentida pela máquina. Esse processo demanda na quebra do programa em várias partes, para que seja entendido sua estrutura e significado. Para isso temos o front-end do compilador que é responsável por vários tipos de análises.

Análise Léxica: Quebra a entrada (o código) em palavras conhecidas como tokens.

Análise Sintática: Analisa a saída do analisador léxico estruturalmente.

Análise Semântica: Cálcula o significado do programa.

Além desses passos o compilador ainda conta com geração de código intermediário, optimização de código e geraçãod de código final como fases do seu processo.

3.1 Analisador Léxico

O analisador léxico é responsável por separar uma sequencia de caracteres que representa o programa fonte em entidades ou tokens, estes que são símbolos básicos da linguagem. Durante a análise léxica, os tokens são classificados como palavras reservadas, identificadores, símbolos especiais, constantes de tipos básicos (int, float), entre outras. Ele lê o texto fonte, carácter por carácter identificando os elementos léxicos da linguagem (identificadores), ignorando comentários, brancos, e includes. Pode-se entende-lo como uma forma de verificar determinado alfabeto e então, quando analisamos uma palavra, podemos definir através da análise léxica se existe ou não algum caracter que não faz parte do alfabeto da linguagem de programação.

Para compilar, devem ser seguidos os comandos:

```
> ocamllex lexico.mll
> ocamlc -c lexico.ml
```

Para utiliza-lo, no terminal digite:

```
> rlwrap ocaml
> #use "carregador.ml";;
```

Pode-se utilizar qualquer código deste documento em C para realizar os devidos testes do analisador, para isso com o ocaml aberto e o carregador carregado faça:

```
> lex "nomedoarquivo.c";;
```

Um exemplo de saída de arquivo C correto pode ser visto abaixo (neste caso o programa utilizado foi o da Listagem 2.5 desde arquivo:

Lexico.tokens list = [Lexico.INTEIRO; Lexico.MAIN; Lexico.APAR; Lexico.FPAR; Lexico.ACHAVE; Lexico.INTEIRO; Lexico.ID "n"; Lexico.PONTOEV; Lexico.ID "n"; Lexico.ATRIB; Lexico.LITINT 0; Lexico.PONTOEV; Lexico.FCHAVE; Lexico.EOF]

Segue abaixo alguns programas C que fazem o analisador ter como saída uma mensagem acusando determinado erro:

Capítulo 4

Referências

LLVM Documentation
Trabalhos de outros alunos do curso