

**Universidade Federal de Campina Grande**

**Centro de Engenharia Elétrica e Informática**

**Departamento de Sistemas e Computação**

**Disciplina: Laboratório de Engenharia de Software**

Descrição do Analisador Léxico

E Sintático

Professora: Francilene Garcia

Alunos:

Adalberto Teixeira

Andrey Menezes

Daniel Lucena

Campina Grande

Outubro de 2010

**Compilador**

Colocando de forma simples, um compilador é um programa que recebe como entrada um programa em uma linguagem de programação e traduz para um programa **equivalente** em outra linguagem, a linguagem objeto. Um papel importante do compilador é relatar quaisquer erros no programa fonte detectados durante esse processo de tradução, esses erros são detectados a partir de regras.

Se o programa objeto for um programa em uma linguagem de máquina executável, poderá ser chamado pelo usuário para processar entradas e produzir saída.

**Motivação e objetivo para o projeto**

**Usuários:** Alunos da Disciplina de LOAC. O nosso publico alvo são os alunos da disciplina de LOAC. Um dos objetivos da disciplina é aprender a manipular um processador. Os alunos aprendem na disciplina a construir um processador com operações básicas, no intuíto de conhecer como é um processador e como o mesmo manipula os dados. Alocando dados em registradores, passando os dados pela ULA, e vendo na prática como o processo de computação ocorre, como os processadores se comunicam com memoria e alocam e desalocam dados da mesma. Serão Beneficiados com esse projeto toda a comunidade do curso de computação, já que a disciplina de LOAC é obrigatória.

Será utilizado como uma ferramenta de auxilio ao ensino do conteúdo da dsiciplina. O mesmo também será usado para fazer demonstrações de como o código gerado por uma linguagem de programação é transformado em código de máquina.

O uso desse processador tem o objetivo de melhorar o entendimento das operações básicas de um processador, e como o mesmo processa as instruções.

**Necessidades do cliente**: Fazer uso de uma linguagem de programação, mostrando como essa linguagem interage com o processador e memória. Fazendo com os alunos da disciplina tenham maior entendimento de como é o funcionamento de uma máquina. As funções e procedimentos internos. Mostrar essa abordagem de forma didática, por isso a simplicidade do projeto. De não utilizar subrotinas, nem funçoes, e usar apenas um tipo de dado (integer). O intuito na verdade é abstrair o funcionamento real de uma máquina. Abstração em nível simplicidade. Porém mostrando o funcionamento real de uma máquina e como o código gerado se torna possível de ser executado.

**Metodologia Utilizada**

1. É feita a analise léxica e sintática do arquivo OCL a ser compilado - esta etapa é feita utilizando-se um analisador léxico/sintático construído através de um gerador de analisador léxico/sintático (LEX/YACC). Nesta etapa é construída uma *árvore de derivação sintática* (AST) para verificação da sintaxe. Esta AST, por sua vez possui anotações para ações semânticas e ações de geração de código, que serão executadas apenas posteriormente. Assim esta etapa produz como resultado uma notificação de erro caso haja algum erro léxico/sintático e/ou um arquivo com mnemônicos de ações semânticas e de ações de geração de código, tais como as respectivas informações para a realização das mesmas;
2. É executado o analisador semântico/gerador de código, utilizando como entrada o arquivo gerado na etapa anterior. Podemos dividir esta etapa nas seguintes sub-etapas:
   1. É lido o arquivo sint\_out.c a partir desse arquivo lemos todo o arquivo compilado e quebrado em tokens. E a partir desses tokens fazemos a geração de código de acordo com a entrada que foi lida.
   2. Para cada declaração de variável, é alocado um espaço em memória para essa variável. Essa memória é uma estrutura de dados manipulada por nosso compilador. Escolhemos como estrutura de dados uma array, por motivos de facilidade.
   3. A partir da gramática de nossa linguagem através da adição de ações semânticas ao longo da tradução das regras geramos uma linguagem intermediaria de processamento.
   4. A partir dos tokens processados fazemos a geração de código para cada expressão de acordo com o operador e ações semânticas colocadas ao longo da análise sintática.

**Analisador Léxico (Definições Regulares)**

**Nome Token Semântica**

Coldelimi [ \t\r] ignora espaço em branco

Linedelimi [\n] ignora quebra de linha

digito [0-9] dígitos 0 a 9

int {digito}+ união de digitos

letra [A-Za-z] letra alfabeto

id {letra}+({letra}|{digito})\* identificador

relop "<"|">"|"=="|"!="|"<="|">=" Operadores Relacionais

logop "&&"|"||" Operadores Lógicos

if IF if

else ELSE else

while WHILE while

do DO do

"{" BEG Inicio de bloco

"}" END fim de bloco

"!" NOT operador negacão

"-" MINUS negação de número

"+" PLUS operador soma

"\*" MULT operador multiplicação

"/" DIV operador divisão

"(" APAREN mod precedencia

")" FPAREN mod precedencia

"=" ATRIB atribuicao

"&" ANDBIT AND bit a bit

"|" ORBIT OR bit a bit

"int" TYPE\_INT tipo int

"main()" MAIN inicio de programa

";" PTVIR fim de comando

":" DPT operador ternário

"?" INTERROG inicio operador terno

**Gramática (Analisador Sintático)**

programa : MAIN BEG comando END

| /\* vazio \*/;

comando : expressao ptvir

| ptvir

| comandoComposto

| comandoIf

| comandoWhile

| comandoDo

;

comandoComposto : comando declaracao

;

declaracao : tipoDeclaracao nomeDecl ptvir

;

tipoDeclaracao : TYPE\_INT

;

nomeDecl : ID {printf("VRI %s\n",$1);}

;

expressao : expressaoAtribuicao

| comando expressaoAtribuicao

;

expressaoAtribuicao: expressaoCondicional

| expressaoUnaria operadorAtribuicao{printf("%s\n",$2);} expressaoCondicional

;

operadorAtribuicao: ATRIB{$$="ATR =";}

;

expressaoCondicional: logicaOuExpressao

;

logicaOuExpressao: expressaoUnaria

| expressaoUnaria LOGOP {printf("%s\n",$2);} expressaoCondicional

| expressaoUnaria RELOP {printf("%s\n",$2);} expressaoCondicional

| expressaoUnaria andOrBit {printf("%s\n",$2);} expressaoCondicional

| expressaoUnaria addOp {printf("%s\n",$2);}expressaoCondicional

| expressaoUnaria multOp {printf("%s\n",$2);}expressaoCondicional

;

expressaoUnaria : literalId {printf("%s\n",$1);}

| literalNum{printf("%s\n",$1);}

| APAREN expressao FPAREN

| unaOp{printf("%s\n",$1);} expressaoUnaria

;

addOp : PLUS {$$="+";} | MINUS {$$="-";}

;

multOp : MULT {$$="\*";} | DIV {$$="/";}

;

unaOp : NOT {$$="!";} | MINUS {$$="-";}

;

andOrBit : ANDBIT {$$="&";} | ORBIT{$$="|";}

;

literalNum : INT {$$=$1;}

;

literalId : ID {$$=$1;}

;

ptvir : PTVIR {$$=";";}

;

comandoIf : IF APAREN expressao FPAREN comando ELSE comando ptvir

| IF APAREN expressao FPAREN comando ptvir

;

comandoWhile: WHILE APAREN expressao FPAREN comando ptvir

;

comandoDo : DO comando WHILE APAREN expressao FPAREN

;

    O analisador sintático tem como objetivo verificar se uma determinada sequência de tokens que lhe é fornecido segue uma determinada estrutura sintática, ou seja, se pertence à linguagem especificada por uma dada gramática. Para isso ele faz uso de uma *árvore de derivação sintática* (AST), que vai sendo construída à medida que os tokens são fornecidos. O uso desta estrutura de dados facilita a detecção de erros sintáticos uma vez que possibilita navegar na estrutura gramatical da linguagem em análise. Entretanto, existem diversos algoritmos para construção desta árvore, que são classificados em descendentes e ascendentes.

**YACC**: o YACC é um programa que faz transformações de código, gerando a árvore de análise sintática. Ele possui compatibilidade e é utilizado em conjunto com o LEX, bastando para isso fazer o include do arquivo LEX, dentro do código fonte do YACC. Ele gera um analisador sintático que utiliza o algoritmo LALR.