

Processamento de Linguagens (3º ano de Curso)

Trabalho Prático 2

Relatório de Desenvolvimento

Jorge Miguel Sol Ferreira (a64293)
Pedro José Freitas da Cunha (a67677)
José Pedro Brito Pereira (a67680)
Grupo 35

6 de Junho de 2015

Resumo

Este relatório documentará todos os passos tomados na realização do segundo trabalho prático da Unidade Curricular de Processamento de Linguagens. Neste projecto é requerida a implementação de um compilador de uma Linguagem de Programação Imperativa Simples e posteriormente gerador de código assembly para uma máquina de stacks virtual.

Conteúdo

1	Introdução	2
2	Análise e Especificação	3
2.1	Descrição informal do problema	3
2.2	Especificação do Requisitos	3
3	Concepção/desenho da Resolução	4
3.1	Gramática	4
3.2	Estruturas de Dados	5
3.3	Exemplos de Programas da Linguagem	6
3.3.1	Programas com erros sintáticos	6
3.3.2	Programas com erros semânticos	6
3.3.3	Programas correctos	7
4	Codificação e Testes	8
4.1	Decisões de Implementação	8
4.2	Testes realizados e Resultados	8
5	Conclusão	12
A	Código do Programa	13

Capítulo 1

Introdução

Enquadramento Um **compilador** é uma peça de software que transforma o código fonte numa dada linguagem de alto nível em instruções que a máquina entenda (Código Máquina). As fases da compilação incluem:

- Análise léxica
- Análise Sintática
- Análise Semântica
- Geração de Código

Conteúdo do documento Neste documento encontrar-se-ão as fases de resolução do problema especificado

Resultados – pontos a evidenciar O resultado do projecto a desenvolver será um gerador de códigoassembly para uma máquina de stacks virtual, partindo de uma Linguagem de Programação Imperativa Simples.

Estrutura do Relatório

No capítulo 2 iremos apresentar o caso de estudo em causa. No capítulo 3 iremos apresentar a estrutura de dados auxiliar à análise semântica e sua utilização no compilador a desenvolver, bem como fazer um esboço do que queremos que seja a nossa linguagem de programação (Gramática Independente do Contexto), para além de alguns exemplos de frases que tenham erros sintáticos, semânticos e frases correctas segundo a nossa especificação. No capítulo 4 iremos apresentar os passos utilizados para geração de código, se possível ou, em alternativa notificação de erro sintático. Finalmente, no capítulo 5 faremos uma apreciação crítica do trabalho realizado e trabalhos futuros. Em anexo iremos colocar o código desenvolvido que permitirá a geração de código máquina.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

Neste projecto é pretendido o desenho de uma Linguagem de Programação Imperativa Simples, para de seguida criar um compilador que gere pseudo-código Assembly de uma Máquina Virtual de Stacks

2.2 Especificação do Requisitos

Os requisitos para o compilador/linguagem a implementar são os seguintes:

- Permitir manusear variáveis do tipo inteiro(escalar ou array) e do tipo string.
- Realizar as seguintes operações:
 - Atribuições de expressões a variáveis.
 - Ler do Standard Input.
 - Escrever para o Standard Output.
- Ciclos(for, while) e instruções Condicionais(if..else).
- Operações Aritméticas, Relacionais e Lógicas sobre inteiros.
- Indexação sobre arrays.
- As declarações de variáveis deverão ser no início do programa.
- Não deverá ser possível realizar redeclarações nem utilizações sem declaração prévia.
- Se não existirem atribuições a uma variável, o valor da mesma deverá ser indefinido

Capítulo 3

Concepção/desenho da Resolução

3.1 Gramática

Ao desenvolver a gramática tentámos fazer com que a mesma ficasse o mais próximo possível do C, funcionando como uma versão simplificada do C. Abaixo encontra-se a gramática desenhada para a Linguagem:

```
Prog ->Declss Instrs
```

```
Declss ->Decls
```

```
Decls ->InitVar  
| Decls InitVar
```

```
InitVar ->INT Var '[' num ']' ';' '  
| INT Var ';' '  
| STRING Var ';' '
```

```
Var ->id
```

```
Instrs ->Instr  
| Instrs Instr
```

```
Instr ->If  
| While  
| For  
| Atr ';' '  
| IO ';' '
```

```

If ->IF '(' Cond ')' '{ Instrs '}' Else

Else ->&
| ELSE '{ Instrs '}'

While -> WHILE '(' Cond ')' '{ Instrs '}'

For ->FOR '(' Atr ';' Cond ';' Atr ')' '{ Instrs '}'

IO ->PRINT Out
| INPUT Var
| INPUT Var '[' Exp ']'

Out ->Exp
| str

Atr ->Var '=' Exp
| Var '[' Exp ']' '=' Exp

Exp ->Termo
| Exp OpA Termo

Termo ->Fator
| Termo OpM Fator

Fator ->Var
| Var '[' Exp ']'
| num
| str
| '(' Exp ')'
| '!' Exp

Cond : Comp
| '(' Cond ')'
| Cond '&'&' Cond
| Cond '|' '|' Cond

Comp : Exp
| Exp OpComp Exp

```

3.2 Estruturas de Dados

Para realizarmos a análise semântica temos uma tabela de Hash para guardar todos os identificadores de variáveis e seus tipos de dados.

A estrutura de dados da tabela de Hash é a seguinte:

```

struct list{
    char *key;
    char *type;
    int init;
    int ind;
    struct list* next;};

struct table{
    int size;
    int elems;
    struct list **list;
};

```

Com a estrutura acima referida podemos então informações sobre uma variável (Identificador, Tipo, Estado de Inicialização e Índice do Registo na Máquina).

3.3 Exemplos de Programas da Linguagem

Abaixo irão estar apresentados vários programas, em que os dois primeiros não cumprem os requisitos sintáticos (3.3.1) e semânticos (3.3.2) da nossa linguagem. Posteriormente iremos apresentar um programa que esteja sintática e semanticamente correcto (3.3.3).

3.3.1 Programas com erros sintáticos

Programa 1:

O programa abaixo quebra uma das regras estipuladas para a Linguagem de Programação: "As declarações de variáveis deverão ser no início do programa."

```

int a;
int b;
for(b=0;b<10;b=b+1)
    int c=b;
print a;

```

3.3.2 Programas com erros semânticos

Programa 1:

O programa que se segue quebra uma das especificações da linguagem: "Não deverá ser possível utilizações de variáveis sem declaração prévia."

```

int a;
int b;

```



```

for(a=0;a<10;a=a+1)
    for(b=10;b>0;b=b-1)
        c=c+a+b;
print a;

```

3.3.3 Programas correctos

Programa 1:

O programa abaixo segue as regras sintáticas e semânticas:

```

int n;
int first;
int second;
int next;
int i;

first=0;
second=1;

print "Insira o numero de termos";
input n;

for(i=0;i<n;i=i+1){
    if(i<=1){
        next=i;
    }
    else{
        next = first + second;
        first = second;
        second = next;
    }
    print next;
}

```

Capítulo 4

Codificação e Testes

4.1 Decisões de Implementação

Ao decorrer do desenvolvimento do gerador de código assembly um dos desafios que foi necessário resolver foi a criação de labels para controlar os saltos que as instruções condicionais necessitam. Para resolver essa situação criámos uma stack que servirá para guardar os números das labels necessárias. Por cada *if sem else* será adicionada uma label, por cada *if com else*, *while*, *for* serão adicionadas duas labels à stack. Por questão de simplicidade criámos um limite de 100 labels a criar.

4.2 Testes realizados e Resultados

Mostram-se a seguir alguns testes feitos (valores introduzidos) e os respectivos resultados obtidos:

Para o exemplo encontrado em 3.3.3 o assembly gerado é o seguinte:

```
PUSHI 0
    PUSHI 0
    PUSHI 0
    PUSHI 0
    START
    PUSHI 0
    STOREG 1
    PUSHI 1
    STOREG 2
    PUSHS "Insira o numero de termos"
    WRITES
    READ
```

```

        ATOI
        STOREG 0
        PUSHI 0
        STOREG 4
L2:      PUSHG 4
        PUSHG 0
        INF
        JZ L1
        JUMP L4
L3:      PUSHG 4
        PUSHI 1
        ADD
        STOREG 4
        JUMP L2
L4:      PUSHG 4
        PUSHI 1
        INFEQ
        JZ L5
        PUSHG 4
        STOREG 3
        JUMP L6
L5:      PUSHG 1
        PUSHG 2
        ADD
        STOREG 3
        PUSHG 2
        STOREG 1
        PUSHG 3
        STOREG 2
L6:      PUSHG 3
        WRITEI
        JUMP L3
L1:      STOP

```

Cálculo do menor elemento de um array: Input:

```

int a[30];
int i;
int num;
int menor;

```

```

print "Insira o numero total de elementos:";
input num;

for(i=0;i<num;i=i+1){
print "Insira um numero";
input a[i];
}

menor = a[0];

for(i=0;i<num;i=i+1){
if(a[i]<menor){
menor = a[i];
}
}

print "O menor elemento é o: ";
print menor;

```

Output:

```

PUSHN 30
PUSHI 0
PUSHI 0
PUSHI 0
START
PUSHS "Insira o numero total de elementos:"
WRITES
READ
ATOI
STOREG 2
PUSHI 0
STOREG 1
L2:
PUSHG 1
PUSHG 2
INF
JZ L1
JUMP L4
L3:
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP L2
L4:
PUSHS "Insira um numero"

```

```

WRITES
PUSHG 1
PUSHG 0
READ
ATOI
STOREN
JUMP L3
L1:
PUSHI 0
PUSHG 0
LOADN
STOREG 3
PUSHI 0
STOREG 1
L6:
PUSHG 1
PUSHG 2
INF
JZ L5
JUMP L8
L7:
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP L6
L8:
PUSHG 1
PUSHG 0
LOADN
PUSHG 3
INF
JZ L9
PUSHG 1
PUSHG 0
LOADN
STOREG 3
L9:
JUMP L7
L5:
PUSHS "0 menor elemento é o: "
WRITES
PUSHG 3
WRITEI
STOP

```

Capítulo 5

Conclusão

Um compilador é uma peça de software complexa que tem uma tarefa crítica na geração de executáveis. Todas as instruções em código máquina deverão estar correctas e deverá haver pouquíssima tolerância a erros, já que a falha em alguma das instruções poderá ter consequências catastróficas.

Acima foi apresentada a gramática, decisões tomadas na geração de código máquina e exemplos de programas correctos segundo a mesma. No apêndice deste documento iremos apresentar o código utilizado na resolução do projecto. Neste momento temos desenvolvido um gerador de código assembly que gera correctamente código máquina, trabalhando de momento com inteiros, arrays de inteiros e Strings. O facto da stack de labels ter uma limitação de 100 labels pode ser limitador para gerar código para programas de grande dimensão. Ainda será preciso adicionar bastantes funcionalidades a este compilador como suporte a funções e reconhecimento de mais tipos de dados.

Para aprimorar os resultados obtidos neste projecto os seguintes pontos deverão ser resolvidos:

- Aumentar a capacidade da stack de labels, ou possivelmente torná-la dinâmica.
- Adicionar o suporte a funções e tipos de dados adicionais.

Apêndice A

Código do Programa

Código do Analisador Léxico para reconhecer os símbolos terminais:

```
%{  
#include <stdlib.h>  
%}  
%x COMENTARIO
```

```
%option yylineno
```

```
num [0-9]+  
pal [a-zA-Z]+
```

```
%%
```

```
\\\/.* {  
;  
}
```

```
"int"      {  
return INT;  
}
```

```
"string"   {  
return STRING;  
}
```

```
"if" {  
return IF;  
}
```

```
"else" {
```

```

return ELSE;
}

"while" {
return WHILE;
}

"for" {
return FOR;
}

"print" {
return PRINT;
}

"input" {
return INPUT;
}

[+-] {
yylval.valOp = yytext[0];
return OpA;
}

[*\/%] {
yylval.valOp = yytext[0];
return OpM;
}

\[^\"]*\[ {yylval.valc = strdup(yytext);return str;BEGIN INITIAL;}

([<>][=]?|"=="|"!=") {
yylval.valc = strdup(yytext);
return OpComp;
}

([<>\(\)\{\}\[\];=!&|'"|"]) {
return yytext[0];
}

{pal} {
yylval.valc = strdup(yytext);
return id;
}

{num} {

```



```

yylval.vali = atoi(yytext);
return num;
}

```

```

.|\\n      {
;
}

```

```

%%

```

```

int yywrap()
{ return(1); }

```

Código do Analisador Sintático/Analisador Semântico/Gerador de Código Máquina:

```

%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hashTable.h"

int countLabel=1;
int labelStack[100], sp=0;
FILE *f;

HashTable symbolTable;
char **bloco;
int i;

void insertSymbol(char* symb, char* type, int tamanho){

int res;
char aux[1000];

res = hashInsert(symbolTable, symb, type, tamanho);

if(res == 0){
sprintf(aux,"Variável '%s' já definida.",symb);
yyerror(aux);
}
}

int checkSymbol(char* symb){

int res;
char aux[1000];

```

```

res = hashContains(symbolTable, symb);

if(res == 0){
    sprintf(aux,"Variável '%s' não definida.",symb);
    yyerror(aux);
}
return res;
}

char* checkType(char* symb){

    int res;

    res = hashContains(symbolTable, symb);

    if(res == 0){
        return "ND";
    }
    else{
        return hashType(symbolTable,symb);
    }
}

void checkSymbolInit(char* symb){

    int res;

    res = hashIsInit(symbolTable, symb);

    // printf("Warning linha %d: Variável '%s' não inicializada!\n",symb);
}

void initSymbol(char* symb){

    hashInit(symbolTable, symb);
}

%}

%union {
    int vali;
    char* valc;
    char valOp;
}

```

```

%token STRING INT IF ELSE WHILE FOR PRINT INPUT

%token <valc> id OpComp str
%token <vali> num
%token <valOp> OpA OpM

%type <valc> Var
%type <vali> Exp Termo Fator

%%

Prog : Declss Instrs {fprintf(f,"\tSTOP\n");}
;

Declss : Decls {fprintf(f,"\tSTART\n");}

Decls : InitVar {}
| Decls InitVar {}
;

InitVar : INT Var '[' num ']' ';' '{
insertSymbol($2,"arrayint",$4);
fprintf(f,"\tPUSHN %d\n", $4);
}

| INT Var ';' '{
insertSymbol($2,"int",0);
fprintf(f,"\tPUSHI 0\n");
}

| STRING Var ';' '{
insertSymbol($2,"string",0);
fprintf(f,"\tPUSHS \"\"\n");
}
;

Var : id {$$ = $1;}
;

Instrs : Instr {}
| Instrs Instr {}
;

Instr : If {}
| While {}
| For {}

```

```

| Atr ';' {}
| IO ';' {}
;

If : IF '(' Cond ')' {
labelStack[sp++] = countLabel++;
fprintf(f, "\tJZ L%d\n", labelStack[sp-1]);
}
'{' Instrs '}' Else
;

Else : {
fprintf(f, "L%d:\n", labelStack[--sp]);
}
| ELSE {
fprintf(f, "\tJUMP L%d\n", countLabel);
fprintf(f, "L%d:\n", labelStack[--sp]);
labelStack[sp++] = countLabel++;
}
'{' Instrs '}' {
fprintf(f, "L%d:\n", labelStack[--sp]);
}
;

While : WHILE {
labelStack[sp++] = countLabel++;
fprintf(f, "L%d:\n", countLabel);
}
'(' Cond ')' {
fprintf(f, "\tJZ L%d\n", labelStack[sp-1]);
labelStack[sp++] = countLabel++;
}
'{' Instrs '}' {
fprintf(f, "\tJUMP L%d\n", labelStack[--sp]);
fprintf(f, "L%d:\n", labelStack[--sp]);
}
;

For : FOR '(' Atr ';' {
labelStack[sp++] = countLabel++;
fprintf(f, "L%d:\n", countLabel);
}
Cond ';' {
fprintf(f, "\tJZ L%d\n", labelStack[sp-1]);
fprintf(f, "\tJUMP L%d\n", countLabel+2);
labelStack[sp++] = countLabel++;
}

```

```

fprintf(f,"L%d:\n",countLabel++);
}
Atr ')' {
fprintf(f,"\tJUMP L%d\n", countLabel-2);
fprintf(f,"L%d:\n",countLabel++);
}
'{' Instrs '}' {
fprintf(f,"\tJUMP L%d\n",labelStack[--sp]+1);
fprintf(f,"L%d:\n",labelStack[--sp]);
}
;

IO : PRINT Out {}
| INPUT Var {
fprintf(f,"\tREAD\n");
fprintf(f,"\tatoi\n");
fprintf(f,"\tSTOREG %d\n", hashInd(symbolTable,$2));
}
;

Out : Exp {
if($1==1){
fprintf(f,"\tWRITEI\n");
}
else{
fprintf(f,"\tWRITES\n");
}
}

| str {
fprintf(f,"\tPUSHS %s\n",$1);
fprintf(f,"\tWRITES\n");
}
;

Atr : Var '=' Exp {
char aux[1000];
if(strcmp(checkType($1),"arrayint")!=0){
if(strcmp(checkType($1),"int")==0){
if($3==1){
if(checkSymbol($1)){
initSymbol($1);
fprintf(f,"\tSTOREG %d\n", hashInd(symbolTable,$1));
}
}
else{

```

```

yyerror("Tipos diferentes");
}
}
else if(strcmp(checkType($1),"string")==0){
if($3==2){
if(checkSymbol($1)){
initSymbol($1);
fprintf(f,"\tSTOREG %d\n", hashInd(symbolTable,$1));
}
}
else{
yyerror("Tipos diferentes");
}
}
else if(strcmp(checkType($1),"arrayint")==0){
if($3==1){
if(checkSymbol($1)){
initSymbol($1);
fprintf(f,"\tLOADN\n");
}
}
else{
yyerror("Tipos diferentes");
}
}
else if(strcmp(checkType($1),"ND")==0){
sprintf(aux,"Variável '%s' não definida.", $1);
yyerror(aux);
}
}
else{
yyerror("Tipos diferentes");
}
}
| Var '[' Exp ']' '=' Exp {
if(strcmp(checkType($1),"arrayint")==0){
fprintf(f,"\tPUSHG %d\n", hashInd(symbolTable,$1));
fprintf(f,"\tSTOREN\n");
}
else{
yyerror("Tipos diferentes");
}
}
;

Exp : Termo {}

```

```

| Exp OpA Termo {
if($1 == 1 && $3 == 1){
switch($2){
case '+':
fprintf(f, "\tADD\n");
break;
case '-':
fprintf(f, "\tSUB\n");
break;
case '|':
fprintf(f, "\tADD\n");
}
}
else if($1 == 2 && $3 == 2){
switch($2){

case '+':
fprintf(f, "\tCONCAT\n");
break;
default:
yyerror("Tipos diferentes");
break;
}
}
else{
yyerror("Tipos diferentes");
}
}
;

Termo : Fator {$$ = $1;}
| Termo OpM Fator {
if($1 == 1 && $3 == 1){
switch($2){

case '/':
fprintf(f, "\tDIV\n");
break;
case '*':
fprintf(f, "\tMUL\n");
break;
case '%':
fprintf(f, "\tMOD\n");
break;
case '&':
fprintf(f, "\tMUL\n");

```

```

break;
}
}
else{
yyerror("Tipos diferentes");
}
}
;

Fator : Var {
if(checkSymbol($1))
checkSymbolInit($1);

if(strcmp(checkType($1),"int")==0){
$$=1;
fprintf(f,"\tPUSHG %d\n", hashInd(symbolTable,$1));
}
else if(strcmp(checkType($1),"string")==0){
$$=2;
fprintf(f,"\tPUSHG %d\n", hashInd(symbolTable,$1));
}
else{
}
}

| Var '[' Exp ']' {
if(strcmp(checkType($1),"arrayint")==0){
$$=1;
printf("%s\n",checkType($1));
fprintf(f,"\tPUSHG %d\n", hashInd(symbolTable,$1));
fprintf(f,"\tLOADN\n");
}
else{
yyerror("Tipos diferentes");
}
}

| num {$$ = 1; fprintf(f,"\tPUSHI %d\n", $1);}
| str {$$ = 2; fprintf(f,"\tPUSHS %s\n", $1);}
| '(' Exp ')' {}
| '!' Exp {}
;

Cond : Comp {}
| '(' Cond ')' {}
| Cond '&' Cond {
fprintf(f,"\tMUL\n");

```



```

}
| Cond '||'|' Cond {
fprintf(f, "\tADD\n");
}
;

Comp : Exp {}
| Exp OpComp Exp {
switch($2[0]){

case '>':
if($2[1] == '='){
fprintf(f, "\tSUPEQ\n");
}
else{
fprintf(f, "\tSUP\n");
}
break;
case '<':
if($2[1] == '='){
fprintf(f, "\tINFEQ\n");
}
else{
fprintf(f, "\tINF\n");
}
break;
case '=':
fprintf(f, "\tEQUAL\n");
break;
case '!':
fprintf(f, "\tEQUAL\n");
fprintf(f, "\tNOT\n");
break;

}
}
;

%%

#include "lex.yy.c"

int yyerror(char *s){
printf("Erro Sintático linha %d: %s\n", yylineno, s);
}

```

```
int main(){

symbolTable = hashCreate(1000);
bloco = malloc(sizeof(char*)*1000);
f = fopen("assembly","w");
yyparse();
return 0;
}
```