Sistemska programska podrška u realnom vremenu 1

Dokumentacija kompajlera

Student: Filip Milošević RA 193/2019

Sadržaj:

Leksička i Sintaksna analiza

Leksička analiza

Leksička analiza je deklarisana klasa u fajlu **LexicalAnalysis.h,** a definisana u **LexicalAnalysis.cpp**. Kao takva služi nam da pročita i generiše listu tokena iz ulaznog **.mavn** fajla. Ovo postižemo pomocu automata sa konačnim brojem stanja.

Automat sa konačnim brojem

Definisan u fajlu **FiniteStateMachine.cpp** sluzi nam da bismo znali i vodili računa u kom se stanju nalazimo tj. da kada čitamo iz fajla karaktere, posto se oni čitaju jedan po jedan mi ne možemo odmah znati koji token imamo, zato imamo automat koji će nam pomoći da validno napravimo odgovarajuće tokene.

Sva moguća stanja su definisana u nizu **stateToTokenTable**.

Matrica automata se sastoji od redova koji predstavljaju sva stanja i od kolona koje predstavljaju svaki moguci karakter koji je podržan.

StateToTokenTable

```
const TokenType FiniteStateMachine::stateToTokenTable[NUM_STATES] = {
    /*state 00*/    T_NO_TYPE,
    /*state 01*/    T_NO_TYPE,
    /*state 02*/    T_NUM,
    /*state 03*/    T_COMMA,
    /*state 04*/    T_L_PARENT,
    /*state 05*/    T_R_PARENT,
    /*state 06*/    T_COL,
    /*state 07*/    T_SEMI_COL,
    /*state 08*/    T_WHITE_SPACE,
    /*state 09*/    T_NO_TYPE,
    /*state 10*/    T_ID,
    /*state 11*/    T_ID,
    /*state 12*/    T_ID,
    /*state 13*/    T_FUNC,
    /*state 14*/    T_ID,
    /*state 15*/    T_ID,
    /*state 16*/    T_MEM,
    /*state 17*/    T_ID,
    /*state 18*/    T_ID,
    /*state 19*/    T_REG,
    /*state 20*/    T_ID,
    /*state 21*/    T_M_ID,
    /*state 22*/    T_R_ID,
    /*state 23*/    T_ID,
    /*state 24*/    T_ID,
    /*state 25*/    T_ADD,
    /*state 26*/    T_ADDI,
    /*state 27*/    T_B,
    /*state 28*/    T_ID,
    /*state 29*/    T_ID,
    /*state 30*/    T_BLTZ,
    /*state 31*/    T_ID,
    /*state 32*/    T_LA,
    /*state 33*/    T_LW,
    /*state 34*/    T_ID,
    /*state 35*/    T_ID,
    /*state 36*/    T_NOP,
    /*state 37*/    T_ID,
    /*state 38*/    T_ID,
    /*state 39*/    T_SUB,
    /*state 40*/    T_SW,
    /*state 41*/    T_ERROR,
    /*state 42*/    T_LI,
    /*state 43*/    T_NO_TYPE,
    /*state 44*/    T_COMMENT,
    /*state 45*/    T_AND,
    /*state 46*/    T_NOT,
    /*state 47*/    T_SRL,
    /*state 48*/    T_SLL,
    /*state 49*/    T_ID, // N
    /*state 50*/    T_ID, // R
    /*state 51*/    T_ID // L
```

Matrica automata

```
/* state 01 */  { 2, 2, 2, 2, 2, 2, 2, 2, 2,23,27,20,20,20,20,20,20,20,20,20,31,21,34,20,20,20,22,37,20,20,20,20,20,20,20, 9, 3, 4, 5, 6, 7, 8, 8, 8,43},     // START_STATE
```

```
/* state 23 */  {20,20,20,20,20,20,20,20,20,20,20,20,20,24,20,20,20,20,20,20,20,20,20,49,20,20,20,20,20,20,20,20,20,20,20,20,20, 0, 0, 0, 0, 0, 0, 0, 0, 0},     // A - T_ID
/* state 24 */  {20,20,20,20,20,20,20,20,20,20,20,20,25,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20, 0, 0, 0, 0, 0, 0, 0, 0, 0},     // D - T_ID
/* state 25 */  {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,26,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20, 0, 0, 0, 0, 0, 0, 0, 0, 0},     // T_ADD
/* state 26 */  {20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20,20, 0, 0, 0, 0, 0, 0, 0, 0, 0},     // T_ADDI
```

Pogledajmo primer odradjenih tokena T_ADD i T_ADDI. Iz START_STATE ako se desi da pročitamo karakter “a” automat će preći u stanje T_ID slova A. Nakon toga ako dodje karakter “d” preći će u T_ID slova D. Ako se opet pojavi slovo “d” preci cemo u stanje T_ADD. Stanje 20 nam je univerzalni T_ID i služi da nam oznaci labelu i prelazimo u to stanje ako dodje bilo koje slovo koje nije “i”.

GRAMATIKA RA 193\2019

Q -> S ; L	S -> _mem mid num	L -> eof	E -> la rid, mid
	S -> _reg rid	L -> Q	E -> bltz rid, id
	S -> _func id		E -> sub rid, rid, rid
	S -> id: E		E -> lw rid, num(rid)
	S -> E		E -> add rid, rid, rid
			E -> b id
			E -> and rid, rid, rid
			E -> not rid, rid
			E -> srl rid, rid, num
			E -> sll rid, rid, num

Sintaksna analiza nam prolazi kroz listu tokena koju smo dobili kao izlaz iz leksicke analze i provera da li je sintaksno dobar kod.

Izbor Instrukcija

U fazi izbora instrukcija ponovo prolazimo kroz listu tokena i pravimo objekte varijabli, instrukcija i labela i smestamo ih u odgovarajuce liste. Razlikujemo dva tipa varijabli. Memorijske varijable i registarske varijable, dok instrukcija ima deset koje se mogu desiti.

Prva faza prolaska je da idemo listom tokena i pravimo varijable dok ne stignemo do tokena koji ima vrednost **_func** i tada prelazimo da drugu fazu.

Druga faza podrazmeva pravljenje instrukcija i labela ako one postoje.Dok pravimo instrukcije vodimo racuna od destination i source registrima.

Analiza životnog veka promenljive

Da bismo izvršili analizu životnog veka potrebno je prvo da odredimo za date instrukcije sve sledece i prethodne instrukcije. To nam ultimativno predstvavlja graf toka našeg programa.

Nakon što smo uspešno napravili graf toka potrebno je da nadjemo koje su promenljive žive na ulazu u instrukciju, a koje su zive na izlazu iz instrukcije. Liste in i out trazimo na sledeći način:

$$out[n] \leftarrow U_{s \in succ[n]} in[s]$$

$$in[n] \leftarrow use[n] \cup (out[n] - def[n])$$

Out definišemo unijom svih in listi sledecih instrukcija
In definisemo kao uniju korišćenih i onih koje su zive na out listi.

Dodela resursa

Pre nego sto dodelimo promenljive registrima moramo da odredimo graf smetnji. On se odredjuje po sledecem algoritmu:

Prolazimo kroz instrukcije i gledamo def varijablu, ako ona postoji u out imace smetnju sa svim ostalim varijablama iz out liste.

```
_func main;
        la              r4,m1;
        lw              r1, 0(r4);
        la              r5,m2;
        lw              r2, 0(r5);
        add             r3, r1, r2;
```

```
***INTERFERENCE GRAPH***

      r1 r2 r3 r4 r5
r1     0  1  0  0  1
r2     1  0  0  0  0
r3     0  0  0  0  0
r4     0  0  0  0  0
r5     1  0  0  0  0
```

Na prethodnim primerima imamo kako bi trebao da izgleda graf smetnji za date instrukcije.

Nakon definisanog grafa smetnji, prolazimo kroz listu varijabli i dodelicemo ih registrima tako da one nemaju smetnju medjusobno.

Generisanje izlaznog fajla

Trivijalna stvar, koristicemo mogucnosti biblioteka <iostream> u kom nam se nalazi output stream i <fstream> u kom nam se nalazi stream za fajlove.

Napravicemo objekat tipa **ofstream**:

ofstream file(“output.s”);

Da bismo upisivali u fajl koristicemo isti princip kao kod standardnog izlaza i pisati **file <<**.

```
//printing to file
ofstream file("output.s");
file << ".globl main" << endl << endl << ".data" << endl;
for (auto it = varsm.begin(); it != varsm.end(); it++) file << (*it)->getName() << ":\t\t.word " << (*it)->getValue() << endl;

file << endl << ".text" << endl;
for (auto it = insts.begin(); it != insts.end(); it++)
{
    for (auto lit = labels.begin(); lit != labels.end(); lit++)
        if ((*lit)->getInst() == (*it)->getPos()) file << (*lit)->getName() << ":" << endl;
    file << (*it)->getShape() << endl;
}

file.close();
```