УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
НОВИ САД
Департман за рачунарство и аутоматику

ПРОЈЕКТНИ ЗАДАТАК

Кандидат: Никола Милосављевић
Број индекса: RA5-2018

Предмет: Системска програмска подршка у реалном времену I
Тема рада: Идеја и функције коришћене у решењу

Ментор рада: др Миодраг Ђукић

Нови Сад, јун, 2020.

САДРЖАЈ

1. Увод

Пројектни задатак се састојао из четири веће целине које представљају - лексичку анализу, синтаксну анализу, анализу животног века и доделу ресурса. Поред тога било је потребно извршити одређене међукораке. Неки од тих међукорака су прављење листе варијабли и инструкција, као и формирање симплификационог стека како бисмо олакшали реализацију кода. Како смо лексичку анализу добили имплементирану она у овом тексту неће бити детаљно обрађена. Идејни процес реашења може се испратити даљим текстом јер свака већа целина има своју функцију која је извршава.

2. Константе

У *Constants.h* хедер фајлу су дефинисане све константе које се користе у пројекту. Већина ће бити накнадно објашњена.

3. Коначна машина стања

Користи се за прављење токена. Машина ради тако што у зависности од тренутног стања „скаче на следеће“ док не наиђе на терминални симбол и тако прави токене. У приложеном коду се види додатак 3 токена T_DIV, T_OR и T_BLEZ које ће касније бити преточене у одговарајуће инструкције. Постоји 52 стања представљено константом NUM_OF_STATES и 47 подржана карактера предстсављена константом NUM_OF_CHARACTERS.

4. Типови

У *Types.h* се налазе све енумерације коришћене у коду.

За функције и лабеле коришћена је FuncOrLabType која може бити FUN_TYPE или LAB_TYPE у зависности од тога шта је у питању. Има и могућност N_TYPE које се додељује као иницијално стање.

За варијабле коришћена је VariableType која може бити REG_VAR или MEM_VAR у зависности да ли су у питању регистарске или меморијске варијабле респективно. Такође, има и NO_TYPE која служи као иницијално стање.

За токене коришћена је TokenType која на већ постојеће има додате T_DIV,T_OR и T_BLEZ по услову задатка.

За инструкције коришћена је InstructionType која на постојеће има додате I_DIV, I_OR и I_BLEZ које представљају добатне мипс команде које ће бити имплементиране.

За регистре коришћена је REGS у којој су дефинисана четири подржана регистра као и иницијално стање.

5. Синтаксна анализа

У овом задатку коршћен је алгоритам са рекурзивним спуштањем обрађен у вежби 7. Овај алгоритам је прилагођен граматици датој у задатку и имплементиране су функције Q, S, L и E.

5.1. Класа SyntaxAnalysis

Класа у себи садржи :

- Поља :
  - LexicalAnalysis& m_lexicalAnalysis
  - bool m_errorFound
  - TokenList::iterator m_tokenIterator
  - Token m_currentToken
- Конструктор :
  - SyntaxAnalysis(LexicalAnalysis& lex)
- Функције :

bool Do() – функција која служи као покретач синтаксне анализе и позове највишу функцију у рекурзивном спуштању. Ако је повратна вредност фунцкије *true* синтаксна анализа је успешно извршена.

void printSyntaxError(Token token) – функција која служи да испише да је дошло до грешке као и токен који је до тога довео.

void eat(TokenType t) – проверава да ли је садашњи токен типа који се очекује по граматици. Ако јесте помери токен на следећи, ако није поставља грешку.

Token getNextToken() – гетер функција коришћена у претходној функцији.

void Q() , ... , ... и E() су функције које су имплементиране по задатој граматици уз додатак да је у E() имплементирано и подржавање за додате инструкције.

6. Креирање променљивих, функција и лабела

Пре свега, треба поменути глобалне променљиве које се овде први пут уводе. Уводе се :

- std::list<Variable*> g_variables – листа свих променљивих које се праве у програму.

- std::list<FuncOrLab*> g_funcOrLabs – листа свих фунцкија и лабела праве се користе у програму. У овом коду се функције и лабеле посматрају као објекти из простог разлога јер је на њих могуће скочити.
- int g_position – представља глобалну позицију које се додељује свсаком објекту. Под глобално, мисли се на ред у улазном фајлу нумерисано од 1.

6.1. Класа FuncOrLab

Класа у себи садржи :

- Поља :
  - FuncOrLabType m_type
  - std::string m_name
  - int m_position
- Конструкторе :
  - FuncOrLab()
  - FuncOrLab(std::string name, int pos)
  - FuncOrLab(FuncOrLabType v, std::string n, int pos)
- Функције :

Сетери :

void setType(FuncOrLabType t)

void setName(std::string s)

void setPosition(int t)

Гетери :

std::string getName()

int getPosition()

FuncOrLabType getType()

Остале функције :

void printer() – функција која служи да штампа сва поља класе.

6.2. Класа Variable

Класа у себи садржи :

- Поља :
  - VariableType m_type
  - std::string m_name
  - int m_position
  - int m_value
  - Regs m_assignment – регистар који ће бити додељен променљивој
  - Int m_pos_ig_mat – представља број колоне и врсте који одговара променљивој. Све меморијске варијабле имају 0 на овом пољу јер оне не улазе у обзир приликом остатка алгоритма.
- Конструкторе :
  - Variable()
  - Variable(std::string name, int pos)
  - Variable(VariableType v, std::string n, int pos, int val, Regs m_assignment)
- Функције :

Сетери :

void setType(VariableType t)

void setName(std::string s)

void setValue(int t)

void setPosition(int t)

void setAssignment(Regs r)

void setPosIgMat(int t)

Гетери :

std::string getName()

VariableType getType()

int getPosIgMat()

Regs getAssignment()

int getPosition()

int getValue()

Остале функције :

void printer() – исписује на стандардни излаз сва поља променљиве.

6.3. Функција createVariablesFuncsAndLabs

Као улазни параметар прима лексику, коју користи само да би преузела листу токена. Идеја је да се једном петљом пролази кроз све токене и да се у зависности од њиховог типа попуњавају глобалне листе које су споменуте у претходном делу. У фор петљи се проверава да ли је тип токена T_MEM и ако јесте прво се проверава да ли је име самог регистра у одговарајућем формату. Ако јесте у одговарајућем формату проверава се да ли је променљива са тим именом већ дефинисана и ако није, она се додаје у листу а у супротном се пријављује грешка.

Ако није T_MEM онда се проверава да ли је у питању регистарска променљива и ако јесте следи се иста процедура као и за меморијску променљиву.

Потом се проверава да ли је у питању тачка-зарез или двотачка из разлога увећања глобалног бројача који говори на којој смо линији кода.

Проверава се и да ли је у питању функција. Ако јесте проверава се да ли је испоштована синтакса давања имена функцијама, да ли почињу малим словом, и ако јесте се она додаје на листу функција и лабела ако тамо већ не постоји.

Последња могућа ситуација је да је у питању лабела која је означена са T_ID. Идеја је била проверити да ли је следећи токен двотачка, јер ако јесте у питању је лабела а ако није прескаче се и тече се даље.

Поред тога у овом делу је имплементирана и функција createRegVars() која попуњава глобални вектор g_regVars само регистарским променљивама јер су у остатку алгоритма оне од већег значаја.

Такође имплементирана је и функција printVariablesFuncsAndLabs() која служи за испис листа променљивих и листу функција и лабела.

7. Креирање инструкција

Предстасвља процес пролажења кроз листу токена и у зависности од тога на који се наиђе извршава се одређена акција. У овом делу је глобална променљива g_position постављена на своје иницијално стање јер се опет пролази кроз листу и ради одржавања конзистентности глобалне позиције. У колико се наиђе на тачку-зарез или двотачку, као и у прошлом делу, глобална позиција се увећава јер нам то сигнализира прелазак у нови ред.

Идеја је да се пролази кроз листу која се користи ради провераве да ли се наишло на било који токен који представља дефинисану резервисану реч за неку инструкцију. Ако је то случај, прави се нови показивач на инструкцију који се потом додаје у глобалну листу инструкција.

7.1. Класа Instruction

Класа у себи садржи :

- Поља :
  - std::string m_instString
  - int m_position
  - InstructionType m_type
  - FuncOrLabs m_funsLabs – ако функција садржи скокове на одређену лабелу или функцију показивачи на те лабеле или функције се чувају у овој листи.
  - Variables m_dst
  - Variables m_src
  - Variables m_use
  - Variables m_def
  - Variables m_in
  - Variables m_out
  - std::list<Instruction*> m_succ
  - std::list<Instruction*> m_pred
- Конструкторе :
  - Instruction()
  - Instruction(int pos, InstructionType type, Variables &dst, Variables &src)
  - Instruction(int pos, InstructionType type)
- Функције :

Сетери :

void setInstString(std::string t)

Гетери :

InstructionType getType()

int getPosition()

std::string getInstText()

Variable* getFromSrc() – коришћена као деструктивно читање из m_src приликом прављења излазног фајла.

FuncOrLab* getFuncOrLab()

Variables getDest()

Variables getSrc()

Variables getDef()

Variables getIn()

Variables getOut()

- ReturnValueVar и ReturnValueIns су две структуре направљене да садрже почетни и крајњи итератор листе варијабли или инструкција респективно. Коришћене су приликом анализе животног века као олакшање.

ReturnValueVar getItUse()

ReturnValueVar getItOut()

ReturnValueVar getItDef()

ReturnValueVar getItIn()

ReturnValueIns getItSucc()

Остале функције :

void pushDst(Variable* t)

void pushSrc(Variable* t)

void pushFuncOrLab(FuncOrLab* t)

void pushSucc(Instruction* t)

void pushPred(Instruction* t)

void pushUse(Variable* t)

void pushDef(Variable* t)

void pushIn(Variable* t)

void pushOut(Variable* t)

void inErase(Variables::iterator t) – коришћено приликом анализе животног века.

void myUnique() – функција која из сортираног скупа података брише дупликате, коришћена приликом анализе животног века.

void mySort() – функција која сортира скуп података, коришћена приликом анализе животног века.

void printer() – функција која штампа сва поља на стандардни излаз.

7.2 Функција createInsList

Као што је назначено у уводу у ово поглавље, функција служи да попуни глобалну листу инструкција. Састоји се из једне петље која у себи има гранање на случајеве. Сви случајеви су обрађени са истом идејом који чине : прављење новог показивача на инструкцију и иницијализовање њене глобалне позиције и типа.

Следећи корак је попуњавање дестинационе и изворне листе саме инструкције које је реализовано на начин који се ослања на успешно извршену синтаксну анализу. Ако је синтаксна анализа завршена успешно можемо да гарантујемо да се редослед токена након почетног ниже на одређен начин што овај алгоритам и користи.

Имплементиране су и помоћне функције finderVar(Token t) и finderFuncOrLab(Token t) које враћају показиваче који се додају у листе у самим инструкцијама. У функцијама је имплементирана и заштита која омогућава излазак из програма ако се користи лабела или променљива која није дефинисана.

Након тога се прави стринг који представља еквивалентну инструкцију у асемблерском коду. У самом стрингу су коришћени термини „ „ и „ „ који ће

приликом финалног превођења служити као места на која се замењују регистри, меморијске променљиве или лабеле.

7.3 Функција fillSuccAndPred

Функција која служи да попуни наследнике и претходнике сваке инструкције. Након креирања листе инструкција оне су већ соритране у самој листи јер су прављене од токена који долазе један за другим из улазног фајла.

Алгоритам попуњавања се ослања на чињеницу да свака инструкција као следећу добија ону која је испред ње, а као претходну она која је иза ње. Изузеци су прва инструкција, која нема претходнике и последња која нема следбенике. Осим тога изузетке представљају и инструкције скока или „бренчовања“.

Имплементирана је додатна функција ... која враћа итератор на инструкцију која ће бити следбеник посматране инструкције. Манипулише тако што прима показивач на функцију или лабелу, и тражи инструкцију која има глобалну позицију за један већу од те функције или лабеле, тј. враћа итератор на прву инструкцију после задате лабеле.

7.4 Функција fillUseAndDef

Једноставна функција која се ослања на чињеницу да „use“ и „def“ листе могу да садрже само регистарске променљиве, јер се меморијске не користе за анализу. Све изворне променљиве се стављају у „use“ док се све променљиве из листе дестинације стављају у „def“.

8. Анализа животног века променљивих

Функција која кореспондира овој секцији је названа livenessAnalysis. Њен задатак је да попуни in и out листе за сваку инструкцију. Алгоритам коришћен за овај део је еквивалентан алгоритму из девете вежбе. Да би се алгоритам извршио коректно, највише се мора проћи кроз исти за сваку инструкцију. Међутим, случај

је да је могуће да имамо две исте итерације и када се то деси алгоритам се завршава.

Алгоритам се састоји из следећих корака објашњених угрубо :

- Кроз листу се пролази обрнутим редоследом, од последњег до првог
- Направе се плитке копије на in и out листе дате инструкције
- У out листу посматране инструкције се убацују све променљиве из in листе њеног следбеника
- У in листу посматране инструкције се убаце све променљиве из њене out листе
- Изврши се сортирање и брисање дупликата из листа
- Из in листе се бришу све променљиве које се налазе у њеној def листи
- У in листу се упишу све променљиве које се налазе у њеној use листи
- Изврши се сортирање и брисање дупликата из листа
- Провери се да ли су плитка копија са почетка и тренутно стање показивача на in и out листе исти

9. Попуњавање графа интерференције

Граф интерференције или граф сметњи јесте матрица сачињена од два стања, __INTERFERENCE__ i __EMPTY__, која означавају да ли између посматраних регистарских променљивих има сметњи или не. Овај корак служи као припрема за наредни корак који предстаља прављење стека симплификације.

У коду је коришћена глобална матрица g_interferenceGraph.

У функцији doIntereferenceGraph() се пролазећи кроз све инструкције попуњава матрица по правилу да сметње постоје између сваке променљиве коју инструкција дефинише и која је жива на њеном излазу. Сходно томе, имплементирана је додатна функција findGPosition(Variable* p) која као повратну вредност има позицију одређене променљиве у листи променљивих. Пошто је матрица интереференције симетрична у односу на главну осу једноставно се попуњава у једном проласку.

Такође имплементирана је фунцкија printIntereferenceGraph() која штампа матрицу интерференције на стандардни излаз.

10. Прављење стека симплификације

Стек симплификације представља последњи корак пре саме доделе ресурса. Ако се стек може направити за одређени број „боја“ тада се и сам програм може извршити са одређеним бројем регистара.

Сегмент прављења стека је организован у функцији doSimplificationG(int degree) која користи глобалну променљиву g_simplificationStack типа stack<Variable*> . Поред тога, функција у себи има :

- bool not_finished – индикатор краја петље која поједностављује граф
- int count – променљива у коју се уписује степен интерференције посматране променљиве
- int k – број боја које имамо на располагању нумерисане од 0 до __REG_NUMBER__ - 1
- int k_temp – копија броја боја која ће служити да се мења и после враћа на иницјалну вредност
- int place_in_regvars – позиција посматране регистарске променљиве у глобалној листи регистарских променљивих
- vector<bool> flags – низ који говори које променљиве су већ „скинуте“ са графа а које нису

Имплементиран алгоритам се огледа у томе што се броји степен интерференције сваке регистарске променљиве и када се нађе прва која је степена као k_temp она се додаје на стек, њен флег у вектору флегова активира и вредности које одговарају њеним колонама у матрици интерференције се подешавају на __CHECKED_INT__ чиме се коначно добија поједностављен граф. Ако ни једна променљива нема задати степен, проверава се за један мањи степен. Такође, ако смо скинули неку регистарску променљиву са графа морамо опет да вратимо k_temp на иницијалну вредност како бисмо проверили да ли је поједностављени граф могуће даље поједноставити. Ако k_temp дође до

вредности -1 значи да је задати интерференциони граф немогуће обојити са degree боја и тада добијамо преливање меморије.

На крају се опет позива функција doInterferenceGraph() како би се опет попунио граф интерференције који ће бити неопходан за следећи корак.

11. Додела ресурса

У овом кораку се од виртуелно неограничено регистара коришћених у програму број смањује на број регистара који постоји на задатој платформи. За овај задатак ограничење је било 4 регистра. Функција која имплементира овај алгоритам се у коду води под именом doResourceAllocation().

Алгоритам представља петља која се извршава докле год има елемената у стеку или док се не искористе сви регистри. У свакој итерацији петље се проверава да ли између текуће регистарске променљиве и неке од већ обрађених променљивих постоји сметња. Ако не постоји, посматраној променљивој се додаје исти регистар као и претходној, а ако постоји, додаје се нови регистар ако их и даље има слободних. Након обраде, показивач на обрађену променљиву се додаје у листу обрађених и наступа следећа итерација.

На крају саме фунцкије имплементирано је исписвање свих регистарских променљивих и њихових ново-додељених регистара.

12. Прављење излазног фајла

Глобалне променљиве које су коришћене за овај део програма су g_output_globl , g_output_data и g_output_text као и g_output_str који представља финални стринг који се уписује у излазни фајл. Прве три споменуте глобалне променљиве су мапе чији је кључ целобројног типа замишљен да буде глобална позиција задате променљиве, функције или лабеле, а вредност је стринг који представља еквивалентну инструкцију. Мапа је коришћена из разлога што се она сама сортира тако да не морамо да бринемо и редоследу излаза јер ће он већ бити сам испоштован на основу глобалне позиције са улаза.

Имплементирана је функција makeFinString(Instruction* i) која од задатог показивача на инструкцију прави стринг који је еквивалентан задатој инструкцији. У њој се пролази кроз инструкциони стринг саме инструкције и „~s“ и „~d“ се мењају одговарајућим терминима.

Осим тога додана је и функција addJr() која додаје „jr $ra“ на крај сваке функције, у њој се користи функција compareFuncOrLab() која служи за поређење функција и лабела по позицији.

На крају, функција printOutputString(string f) користи се да направи коначни стринг од горе наведене три мапе и да тај стринг испише на стандардни излаз као и у фајл чије се име шаље као параметар функцији. Добијени излазни фајл се чува у директоријуму „outputs“ и биће сачуван под истим именом као и улазна датотека са екстензијом .s.

13. Тестирање

Програм је подвргаван тестирању уз помоћ достављених улазих датотека, „simple“ и „multiply“ чији се излази налазе у фолдеру „outputs“ . Треба напомеути да за улазни фајл „multiply“ није могуће направити стек поједностављења по условима задатка јер се његов граф не може обојити са 4 боје. Излазни фајл који је достављен релазиован је уз измену кода тако да постоји 5 регистара.