Project Report

Abstract

This is the report for CS215 project #2, which is divided into following part:
I. Contents. Specify the files of package.
II. Basics. Illustrate the foundations of project.
III. Frameworks. Principles, function list & symbol table.
IV. Key points. Some crucial points to deal with.
V. Optimizations. Semantic analysis & register allocation.
VI. Extensions. Syntax tree visualization DIY.
VII. Acknowledgement. Thanks for everyone aiding me in the project.
Appendix. DOT language parametres reference.

I. Contents

5110309193.rar
- simplified //of the request, `./scc in out`
    - smallC.l //flex file
    - smallC.y //bison file
    - makefile
- ultimate //beyond the request, `./scc in out1 out2 out3`
    - smallC.l //flex file
    - smallC.y //bison file
    - makefile
    - DOT gallery
- report.pdf

II. Basics

In this part, I will cover some foundations of my work. For such a large project, you need to think a lot before coding. So, how do I think, precisely?

Following the guide document given by TA, it's a piece of cake for me to install LLVM-3.3. After that, I skipped the language manual in llvm.org, as well as try to run a few toy Small-C programs. Clearly that LLVM IR bears a striking resemblance to assembly language. In order to manipulate it, we need to manupulate registers first.

These are all I knew before the distribution of test cases. Anyway, the test cases opened my eyes. As long as you rewrite the Small-C test cases into standard C ones, `./clang -emit-llvm -S *.c -o *.s`, and carefully read the LLVM IR generated, surely you will know how the language works.

Totally there are 3 core conceptions, lable, address and register. Lable is for jump. Address, as divide to global(@) and local(%), is used for load/store. And for register, intermediate calculation and load/store are its main jobs.

With a clear understanding of LLVM IR grammers, I turned to the grammar of Small-C. From the internet, I got the info. that LLVM IR are generated by AST(Abstract Syntax Tre) traversal, which is done by its header files. And now, all I have is the syntax tree obtained in the first step of project. As a result, I need to analyze the grammar of Small-C now.

To tell the truth, I started the coding work after only 50% understanding of Small-C grammar. After all, what we need to add is nothing but semantic actions when traversing the tree... That sounds easy. So, don't be hesitate. Let the traverse begin.

III. Frameworks

I have to say that, the task is more difficult than I've imagined. Well, should say complicated much more precisely...You have to be as careful as a girl. Making a variety of stupid mistakes, I successfully accomplished the first 6 test cases in 13/12/19, with 3 sleepless days.

Instead of bottom-up approuch(which is more powerful cause it need only one pass), I chose top-down one. it seems that the latter is more explicit. The followings are my frameworks. First, let's talk about principles:

1. Simple is better than complex.
2. Special case, special judge.
3. Good in local means good in whole.

Using the principles metioned previously, I finished the code within 1k5 lines. Well, half quantity of some students' work, but the same quanlity.

Function Lists:

```
void _Program(struct treeNode* root);

void _Extdefs(struct treeNode* t);

void _Extdef(struct treeNode* t);
void _ExtdefStruct(struct treeNode* t);
void _ExtdefStrId(struct treeNode* t);
void _ExtdefStrOp(struct treeNode* t);

void _ExtvarsType(struct treeNode* t);
```

```
void _ExtvarsStrId(struct treeNode* t);

void _DecExt(struct treeNode* t);
void _DecStrId(struct treeNode* t);
void _Decs(struct treeNode* t);
void _DecInner(struct treeNode* t);

void _ArgsExt(struct treeNode* t);
void _ArgsInner(struct treeNode* t);
void _ArgsFunc(struct treeNode* t);

void _Defs(struct treeNode* t);
void _Def(struct treeNode* t);
void _DefsStrOp(struct treeNode* t);
void _DefStrOp(struct treeNode* t);

void _Func(struct treeNode* t);
void _Paras(struct treeNode* t);
void _Para(struct treeNode* t);
void _Stmtblock(struct treeNode* t);
void _Stmts(struct treeNode* t);
void _Stmt(struct treeNode* t);
char* _Exp(struct treeNode* t);
```

And let me explain the principles shown in the frameworks:

1. Simple is better than complex.
I'd like to cite _DecExt() and _DecInner() as example. The former is used for declarations of global variables, and the latter is used for that of local variables. Same as it seems, they differ a lot in grammer actually. In this case, to simplify the coding work, seperate them into different non-terminal symbols is a wise choice.

2. Special case, special judge.
Let's talk about struct, which is what I handled at last. Struct seem so special to me at the very beginning, so I abandon the case, focus on the rest of project. After the success of the first 6 cases, I've accumulated enough exp. to deal with struct case. At that moment, 3 hours is all I need to code struct.

3. Good in local means good in whole.
The only function which returns char*, _Exp(), is my favourite as well. I use loadFlag to control its load or not, while returning register or INT/address of ID. The elegance of _Exp() benifits the whole projects. See, op1 = _Exp(*), op2 = _Exp(*), result = op op1, op2, and we return result. Recursion and pointers are two spectacular tools in the

process, and the more you use them, the more you will find that how beautiful the CS is, as always.

Talking about the algorithm part of project, DFS+trace, that's all. What about the data structure? Well, I use a symbol Table indexed by the first letter of ID. That is, A/a->symTable[0][*], B/b->symTable[1][*], ..._->symTable[26][*] and it ends. As is know to all, anytime you want to index some strings without rules, hash table is undoubtedly the best choice. Nevertheless, it's complicated to write a close hash table. Remembering 'Simple is better than complex', let's just simplify the data structure to an bi-dimension array. And, I will clearly state the structure for you.

```
struct symbol
{
    char* word;              //name of ID
    char type;               //a(.addr), g(@) or l(%)
    char* arrSize;           //record the size of array
    char* structName;        //the corresponding struct name
    int structMem;           //the dimension of struct member
};
```

The data structure is not the best, but good enough in this project. You can use it to deal with anything in the field of LLVM IR. Yes, KISS(Keep It Simple, Stupid) is also my favourite.

IV. Key points

The LLVM IR do 3 things in 80% of its time. That is, load, store and op op1,op2. And, I use 20% of my coding time dealing with it...Another concrete example of 80/20 principles. So, where did I put the rest 80% of time? In those confusing key points, surely.

1. Parametres

Q: int dfs(int x), x is defined in PARAS function. However, it needs to be loaded when entering the STMTBLOCK. How to deal with the case?

A: Well, int paraFlag(to denote whether we have parametres), int paraPoint(how many parametres), char* paraArr[10](store the value of paramtres).

2. {} of STMTBLOCK

Q: How do we know when to print { and } encountering with STMTBLOCK?

A: It's easy, we maintain a variable called entryDepth. ++it when it goes deeper into STMTBLOCK, --it when gets out. Only when it is zero should we print { and }.

3. Load or not?

Q: When to load, and when not?
A: We load those which are not left values(in the left of ASSIGNOP). Define loadFlag in EXP will facilitate a lot, which means few copies of same codes. Meeting with left values, all we need to do is loadFlag = 0, then loadFlag = 1.

4. Multiple variables of array and struct

Q: How to deal with the case?
A: Well, almost the same as parametres. I think you got it, do you?

5. bitcast

Q: What the hell is bitcast? int a[2] = {1,2} seems a difficult job now...
A: Don' care about bitcast. Why not rewrite int a[2] = {1,2} into int a[2], a[0] = 1, a[1] = 2? Can you see bitcast anymore this time?

6. Decimal Transformation

Q: 0xD7 and -0327 cases, how to deal with them?
A: I don't know either, until strtol() appears in my sight.

There are a large number of small points as well. But, let's just stop here.

V. Optimizations

1. Semantic Analysis

What I've done is nothing but the same as project 1. When encountering the big number, I will add (overflow!!!) symbol after that.

2. Register Allocation

I wonder if anyone could implement Ershov algorithm in slides of Doc. Wu. As for me, the clock is ticking. So, I just recount the register everytime it goes into FUNC.

VI. Extensions

So much for the basic part. Now, let's talk about my new extensions to syntax tree visualization, which is truly interesting. From now on, you can customize your own syntax tree!

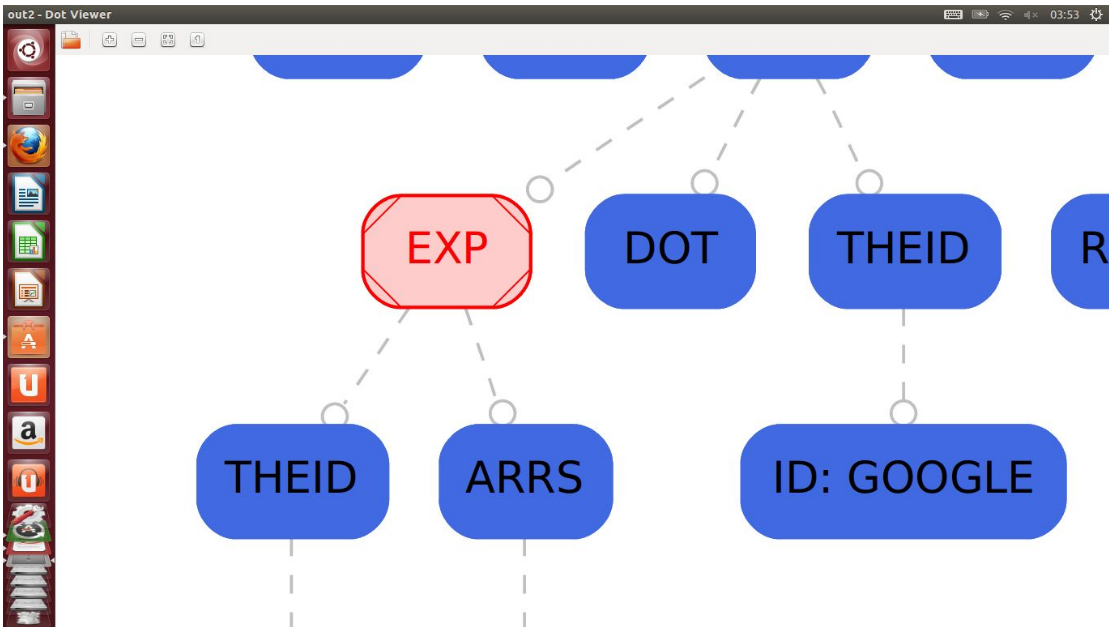
Only the ultimate version contains the cool function, which is as follows:

```
Welcome to the world of Small-C!
First, let's do some preparing work.
Would you like to DIY your syntax tree visualization(y/n)?
y
```

```
OK. Let's move on!
Totally there are 8 steps:
(1/8) Enter the shape of node: record
(2/8) Would you like to round your node(y/n)? y
(3/8) Enter the color of node: royalblue
(4/8) Would you like to fill your node(y/n)? y
(5/8) Enter the periphery of node: 2
(6/8) Enter the style of edge: dashed
(7/8) Enter the color of edge: grey
(8/8) Enter the arrowhead of edge: odot

Enjoy your own figure now!
```

And here is the figure:



The extension work is based on my learning of DOT language. You see, everyone will know how to play it so long as the example is given. The work is interesting and natural, which fits my aethetics. For what parametres can we enter, see Appendix.

VII. Acknowledgement

Thanks to my classmate, GAN Zhenye, who enlightened me the idea of 'loadFlag'. Also for his flushing Renren states about the project, which pushed me a lot.
Special thanks to my dearest TA, WANG Yang, who replies mail in speed of light, and illustrates everything clearly. Thanks for his kindness, fairness and wisdom.
I'm so fortunate to have so many brilliant guys around me. Therefore, I will push myself harder, and harder, and harder...

Appendix

DOT Language Parametres reference:

For this part, I will directly use the data from Drawing graphs with DOT, written by Emden R. Gansner and Eleftherios Koutsofios and Stephen North. And, here are some of my suggestions:

1. Round does not work on all shapes. However, the followings are proved to be work: record, diamond, square and circle.
2. For style of edge, dashed and solid are all I've known.
3. If you are really interested in this, see the following websites:

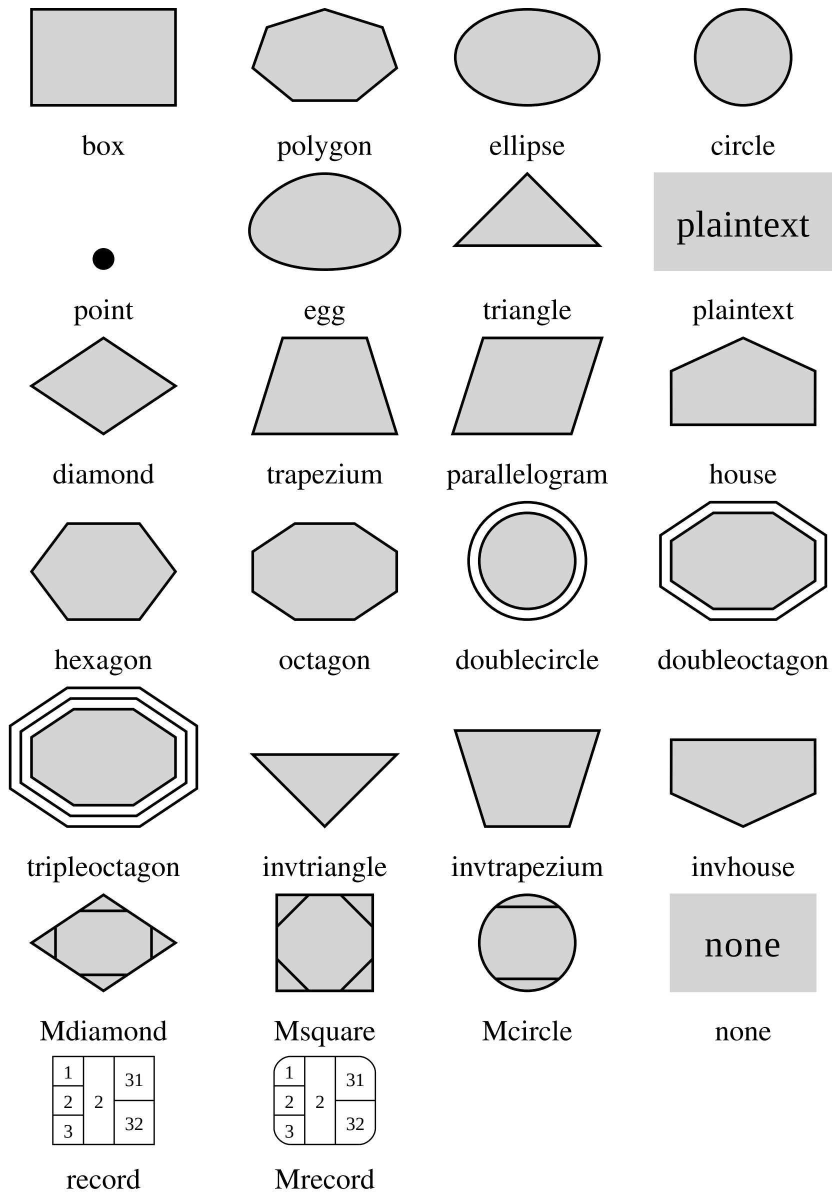
`http://www.graphviz.org/Documentation.php`
`http://www.graphviz.org/Documentation/dotguide.pdf`

Thank you!
WANG Tianze
CS.SJTU
13.12.25

H Node Shapes

These are the principal node shapes. A more complete description of node shapes can be found at the web site

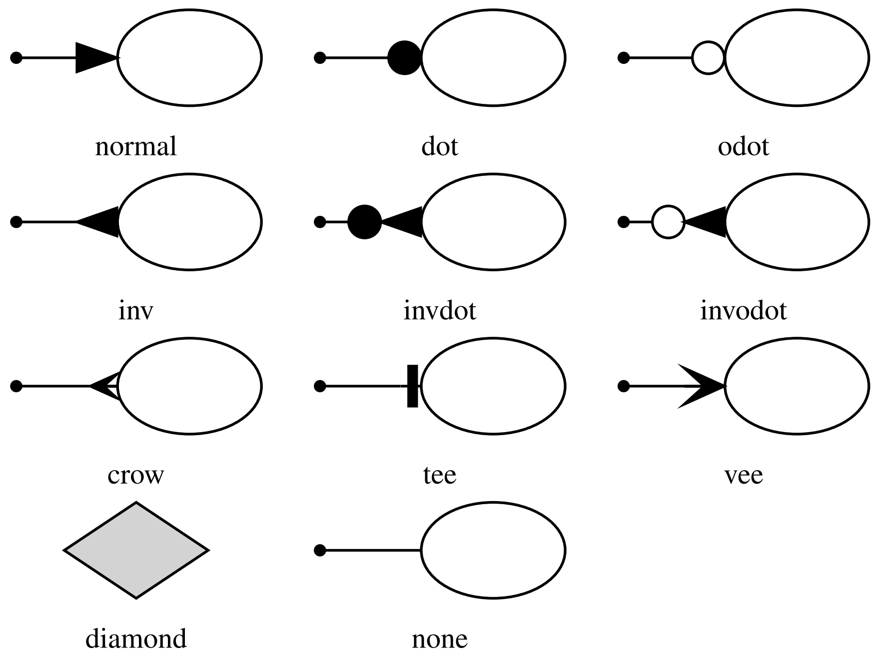
`www.graphviz.org/doc/info/shapes.html`



I Arrowhead Types

These are some of the main arrowhead types. A more complete description of these shapes can be found at the web site

`www.graphviz.org/doc/info/arrows.html`



J Color Names

Here are some basic color names. More information about colors can be found at

```
www.graphviz.org/doc/info/colors.html
www.graphviz.org/doc/info/attrs.html#k:color
```

Whites
antiquewhite[1-4]
azure[1-4]
bisque[1-4]
blanchedalmond
cornsilk[1-4]
floralwhite
gainsboro
ghostwhite
honeydew[1-4]
ivory[1-4]
lavender
lavenderblush[1-4]
lemonchiffon[1-4]
linen
mintcream
mistyrose[1-4]
moccasin
navajowhite[1-4]
oldlace
papayawhip
peachpuff[1-4]
seashell[1-4]
snow[1-4]
thistle[1-4]
wheat[1-4]
white
whitesmoke

Greys
darkslategray[1-4]
dimgray
gray
gray[0-100]
lightgray
lightslategray
slategray[1-4]

Blacks
black

Reds
coral[1-4]
crimson
darksalmon
deeppink[1-4]
firebrick[1-4]
hotpink[1-4]
indianred[1-4]
lightpink[1-4]
lightsalmon[1-4]
maroon[1-4]
mediumvioletred
orangered[1-4]
palevioletred[1-4]
pink[1-4]
red[1-4]
salmon[1-4]
tomato[1-4]
violetred[1-4]

Browns
beige
brown[1-4]
burlywood[1-4]
chocolate[1-4]
darkkhaki
khaki[1-4]
peru
rosybrown[1-4]
saddlebrown
sandybrown
sienna[1-4]
tan[1-4]

Oranges
darkorange[1-4]
orange[1-4]
orangered[1-4]

Yellows
darkgoldenrod[1-4]
gold[1-4]
goldenrod[1-4]
greenyellow
lightgoldenrod[1-4]
lightgoldenrodyellow
lightyellow[1-4]
palegoldenrod
yellow[1-4]
yellowgreen

Greens
chartreuse[1-4]
darkgreen
darkolivegreen[1-4]
darkseagreen[1-4]
forestgreen
green[1-4]
greenyellow
lawngreen
lightseagreen
limegreen
mediumseagreen
mediumspringgreen
mintcream
olivedrab[1-4]
palegreen[1-4]
seagreen[1-4]
springgreen[1-4]
yellowgreen

Cyans
aquamarine[1-4]
cyan[1-4]
darkturquoise
lightcyan[1-4]
mediumaquamarine
mediumturquoise
paleturquoise[1-4]
turquoise[1-4]

Blues
aliceblue
blue[1-4]
blueviolet
cadetblue[1-4]
cornflowerblue
darkslateblue
deepskyblue[1-4]
dodgerblue[1-4]
indigo
lightblue[1-4]
lightskyblue[1-4]
lightslateblue[1-4]
mediumblue
mediumslateblue
midnightblue
navy
navyblue
powderblue
royalblue[1-4]
skyblue[1-4]
slateblue[1-4]
steelblue[1-4]

Magentas
blueviolet
darkorchid[1-4]
darkviolet
magenta[1-4]
mediumorchid[1-4]
mediumpurple[1-4]
mediumvioletred
orchid[1-4]
palevioletred[1-4]
plum[1-4]
purple[1-4]
violet
violetred[1-4]