02242 Program Analysis
Project: A Program Analysis Module

Fariha Nazmul (s094747)
Group 07

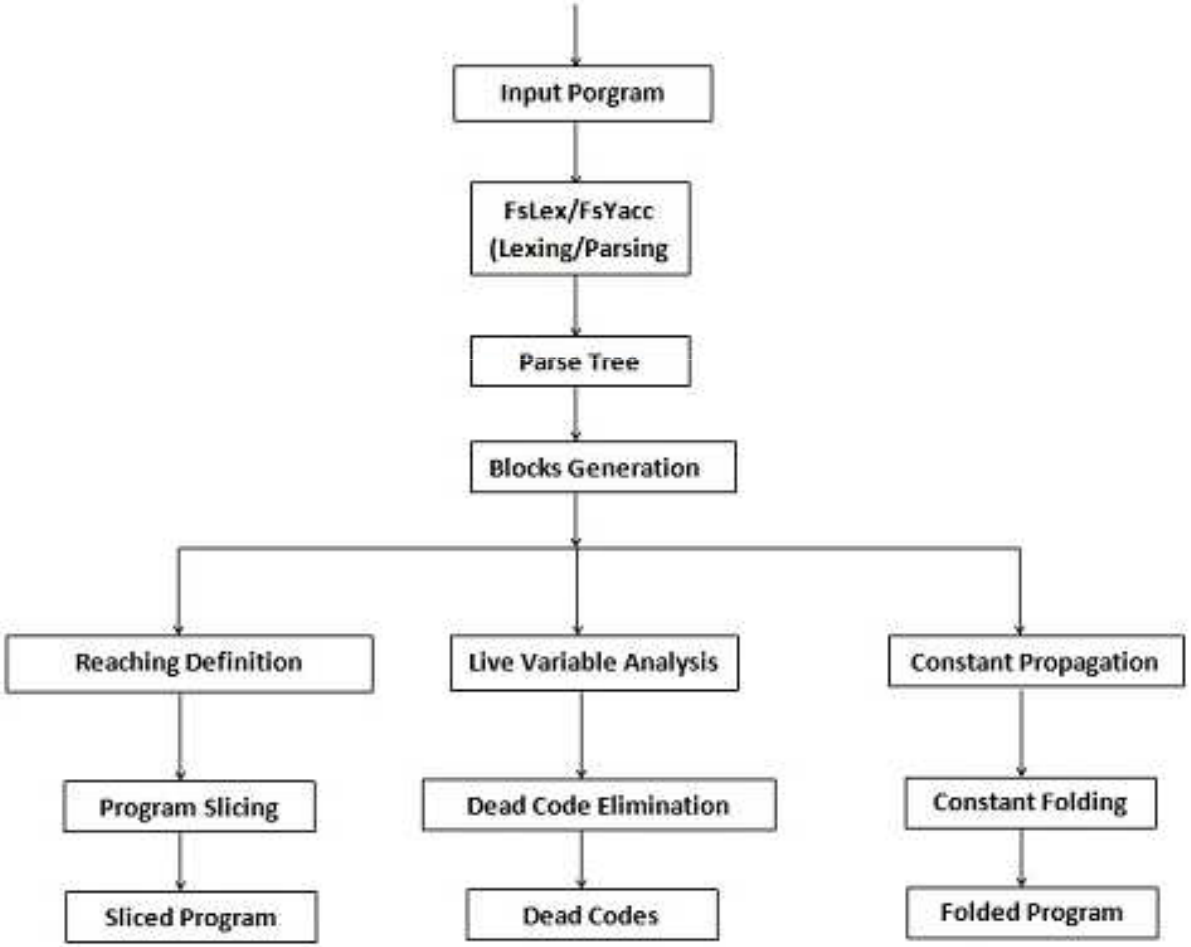
Project Description

- To design and construct a program analysis module for guarded command language and perform a series of experiments with it.

- The analysis module addresses three different analysis problems
  - Program Slicing:
    - To determine the part of the given program that may influence the values computed at a given point of interest.
  - Dead Code Elimination:
    - To determine the part of the given program that does not affect the overall result computed by the program.
  - Constant Folding:
    - To replace the sub expressions of the given program by their values whenever they can be determined at compile time.

Implementation Structure

- The overall design of our project is shown in the following diagram:

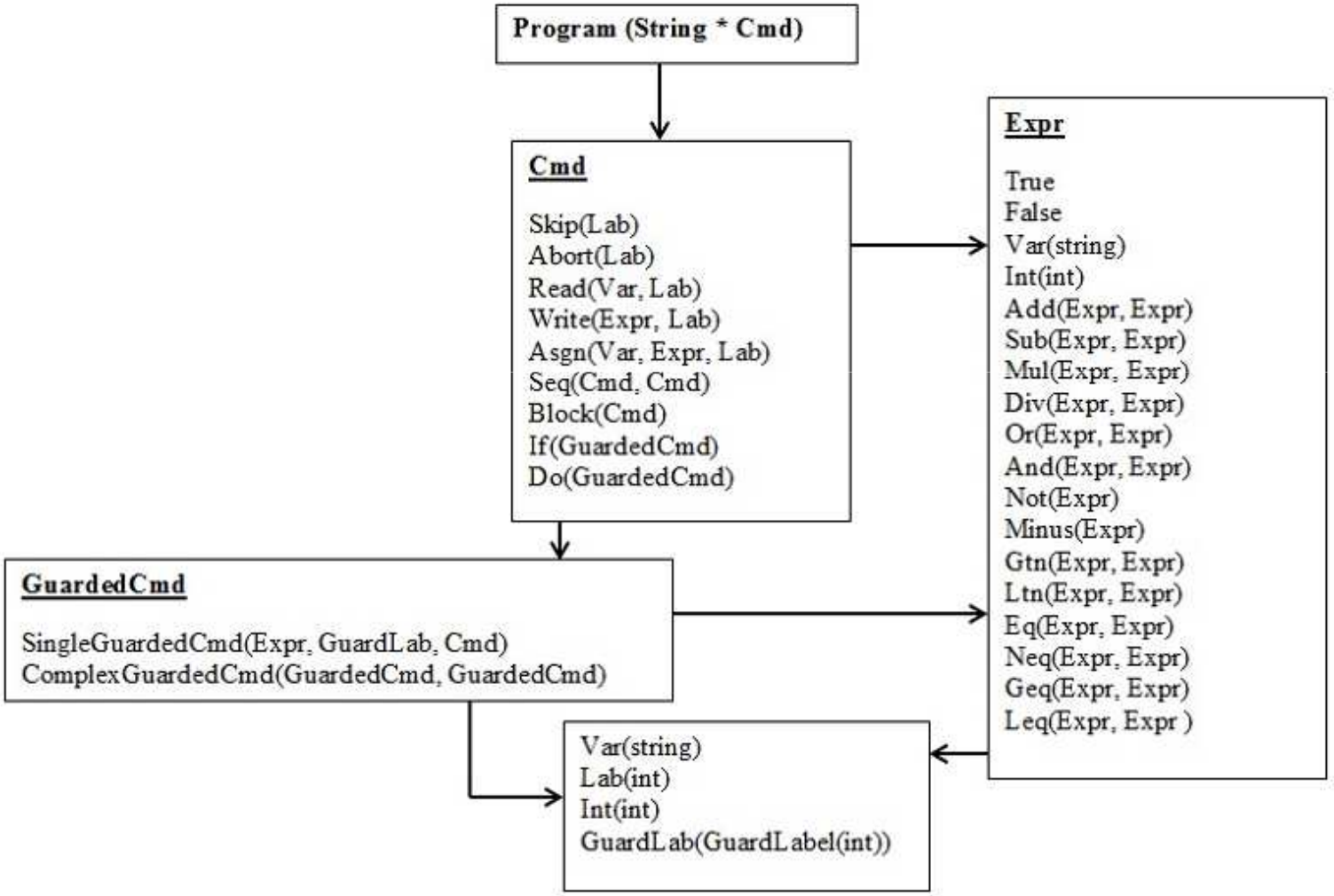


Guarded Command Language

- The syntax of the Guarded Command Language can be represented as below:

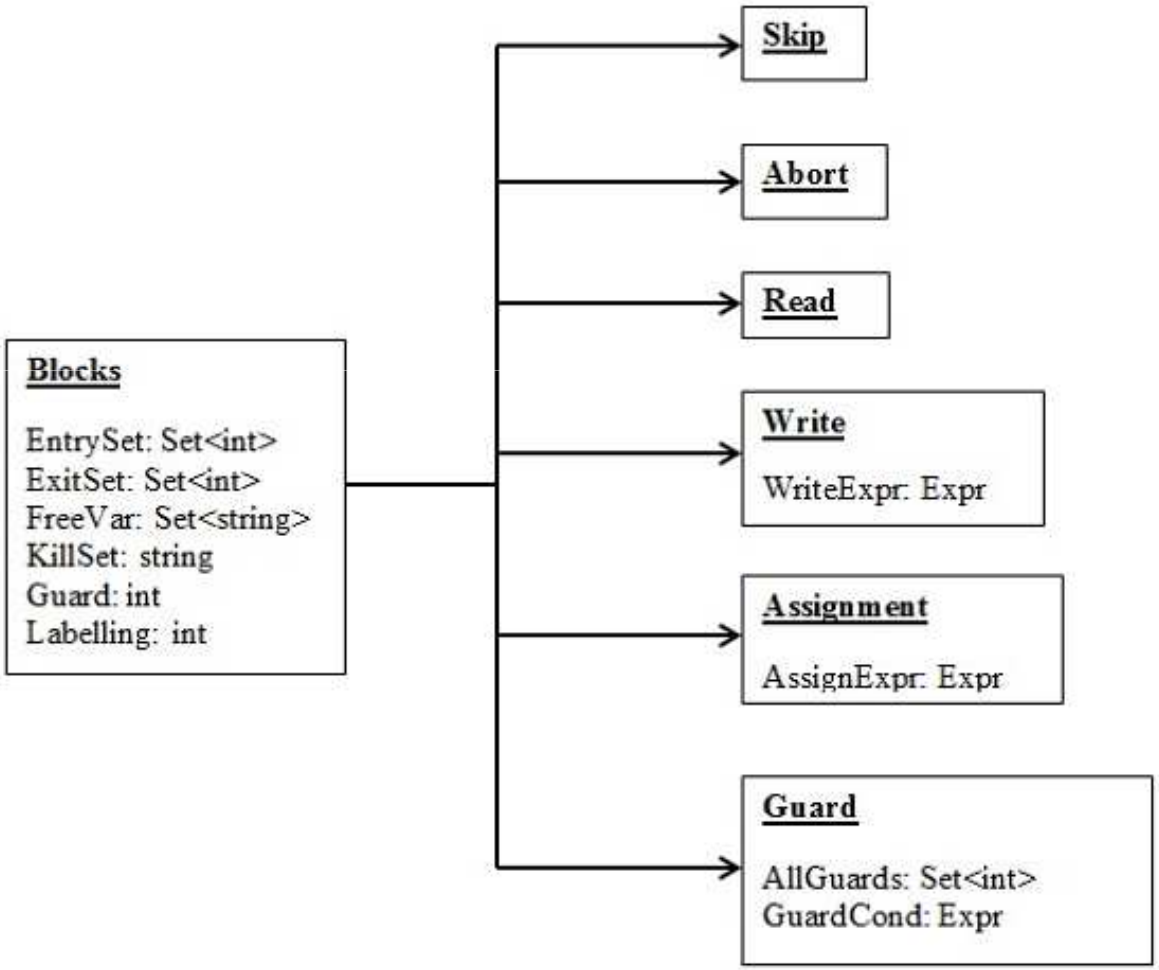
```
Expressions: e ::= n | true | false | x
                 | e1 opb e2 | opm e | (e)

Commands: C ::= x := e | skip | abort
              | read x | write e | C1; C2
              | {C} | if gC fi | do gC od

Guarded commands gC ::= e -> C | gC1 [] gC2

Programs: P ::= module name : C end

opb  {+, -, *, /, int  int  int
      <, >, <=, >=, =, !=, int  int  bool
      &, |} bool  bool  bool

opm  {!, bool  bool -} int  int
```

Parse Tree of GC Language



Data Structure

- We have used the following data structures in our project:



Worklist Algorithm for MFP

- The worklist algorithm used for MFP is given below:

algorithmMFPusingWorklist

1. $workList \leftarrow \emptyset$;
2. *for all* (l,l') *in* F *do*
3. $\quad workList \leftarrow (l,l') :: workList$;
4. *endfor*
5. *for all* l *in* E *do*
6. $\quad$ *if* $(l \in E)$ *then*
7. $\quad\quad Analysis[l] = i$;
8. $\quad$ *else*
9. $\quad\quad Analysis[l] = nil$;
10. *endfor*
11. *while* $(workList \neq nil)$ *do*
12. $\quad (l,l') \leftarrow head(workList)$;
13. $\quad workList \leftarrow tail(workList)$;
14. $\quad$ *if* $f(Analysis[l]) \not\sqsubseteq Analysis[l']$ *then*
15. $\quad\quad Analysis[l'] = Analysis[l'] \sqcup f(Analysis[l])$;
16. $\quad\quad$ *for all* l'' *with* (l',l'') *in* F *do*
17. $\quad\quad\quad workList \leftarrow (l',l'') :: workList$;
18. $\quad\quad$ *endfor*
19. $\quad$ *endif*
20. *for all* l *in* E *do*
21. $\quad MFP_o(l) = Analysis[l]$;
22. $\quad MFP_\bullet(l) = f(Analysis[l])$;
23. *endfor*

Program Slicing Algorithm

- Initialization Phase:

```
algorithmProgramSlicing (programBlocks, labelPOI)
1.  (RDcircle, RDbullet) ← ReachingDefinitions(prgoramBlocks);
2.  programSlice ← ∅;
3.  workList ← ∅;
4.  allGuards ← getAllGuards(labelPOI);
5.  if (allGuards ≠ nil) then
6.        for each (g ∈ allGuards) do
7.              workList ← g ::workList;
8.        endfor
9.  else
10.       workList ← labelPOI ::[ ];
11. endif
```

Program Slicing Algorithm (cont.)

- Iteration Phase:

```
12. while workList ≠ nil do
13.     currentLabel ← head(workList);
14.     workList ← tail(workList);
15.     programSlice ← programSlice ∪ {currentLabel};
16.     guardLabel ← ifUnderGuard(currentLabel);
17.     if (guardLabel ≠ 0) then
18.         allGuards ← getAllGuards(guardLabel);
19.         for each (g ∈ allGuards) do
20.             if ((g ≠ 0) and (g ∉ workList) and (g ∉ programSlice))then
21.                 workList ← g ::workList;
22.             endif
23.         endfor
24.     endif
25.     for each (x ∈ freeVariables(currentLabel)) do
26.         for each ( (x, lPrime) ∈ RDcircle(currentLabel)) do
27.             if ( (lprime ≠ 0)and (lprime ∉ workList)
28.                 and (lprime ∉ programSlice)) then
29.                     workList ← lPrime : : workList;
30.             endif
31.         endfor
32.     endfor
33. endwhile
34. return programSlice
```

Dead Code Elimination Algorithm

algorithmDeadCodeElimination(programBlocks, deadCodeLabels)

```
1.  programCopy ← programBlocks;
2.  (LVcircle, LVbullet) ← LiveVariables(programCopy);
3.  tempDeadCodes ← ∅;
4.  for each label ∈ allLabels(programCopy)do
5.      if ((killLV(label) ∉ LVcircle(label)) and (label ∉ deadCodeLabels)) then
6.          tempDeadCodes ← tempDeadCodes ∪ killLV(label);
7.      endif
8.  endfor
9.  deadCodeLabels ← deadCodeLabels ∪ tempDeadCodes;
10. if (tempDeadCodes ≠ ∅)then
11.     for each label ∈ tempDeadCodes do
12.         programCopy[label] ← skip;
13.     endfor
14.     return algorithmDeadCodeElimination(programCopy, deadCodeLabels);
15. endif
16. return deadCodeLabels;
```

Constant Folding Algorithm

```
algorithmConstantFolding (programBlocks)
1.  (CPcircle, CPbullet) ← ConstantPropagation(programBlocks);
2.  foldedProgram ← programBlocks;
3.  for each label ∈ allLabels(foldedProgram)do
4.        currentCPCircle = CPcircle(label);
5.        listOfConstants ← getConstants(currentCPCircle);
6.        currentBlock ← foldedProgram(label);
7.        if ((currentBlock = Assign)or (currentBlock = Guard) or
8.           (currentBlock = Write)) then
9.                currentExp ← getExpression(currentBlock);
10.               modifiedExp ← substituteConstants(currentExp,listOfConstants);
11.               if(freeVariables(modifiedExp) = ∅)then
12.                       currentExp ← evaluateExp(modifiedExp);
13.               else
14.                       currentExp ← modifiedExp;
15.               endif
16.               setExpression(foldedProgram(label), currentExp);
17.       endif
18. endfor
19. return foldedProgram;
```

Conclusion

- Designed the algorithms for the modules of the project.
- Successfully implemented all the modules of the project problem in F# using Visual Studio 2010.
- Can perform one of the following procedures at a time based on the user input : Programslicing, DeadCodeElimination or ConstantFolding.
- Tested the correctness of our application based on several input programs written in GC Languages.
- Tested our project using the benchmark programs provided by other groups on the Campusnet.

Thank You.