The limitations of CFG

- Given the following grammar

  S ::= Decl Stmt

  Decl ::= Type ***id*** | Decl; Decl

  Type ::= ***string*** | ***int***

  Stmt ::= Stmt; Stmt |

  id = Exp | …

  Exp ::= Exp * Exp | ***id*** | ***num*** | ***char**** | …

- Does the corresponding parser accept the following programs?

```
string x;
int  z;

x= “hello world”;
z=x+1;
```

```
int x;
int  z;

z=x+1;
```

```
int x;

x=0;
z=10/x;
```

Limitations (continued)

- Many other things can not be decided by syntax analysis
  - Does the dimension of a reference match the declaration?
  - Is an array access out of bound?
  - Where should a variable be stored (head, stack,…)
  - …

Semantics Analysis

- The reason of the limitations is that answering those questions depends on values instead of syntax.
- We need to analyze program semantics.
  - Usually, this is done by traversing/analyzing program representations.
    - Examples of representations: AST, Control flow graph (CFG), Program dependence graph (PDG), SSA (single static assignment).
    - Sample semantic analysis: type checking, code generation, register allocation, dead code elimination, etc.

Type Checking

- An important phase in compilation. The goal is to reduce runtime errors.
  - More specifically, we want to check that each expression has a correct type.
- Concepts
  - Symbol tables (environments)
    - We need to look up the declaration of a variable when we encounter it during type checking.
  - Bindings
  - Scope
  - Definition/ use
- Two sub-phases
  - Symbol table construction
  - Type checking

Symbol Tables and Scopes

```
1   public class E {
2       public static int a = 5;
3   }
4   class N {
5       public static int b = 10;
6       private static int a = E.a + D.d;
7       public int foo(int p, int bb) {
8           String a;
9           …
10      }
12       public boolean  bar(int p) {
13          return false;
14       }
15  }
16  class D {
17      public static int d = E.a + N.a;
18      public int foo ( ) {
    }
```

We have:

(a) A global symbol table for forward references.

(b) When type checking a class, we extend the symbol table to class level.

(c) When type checking a method in the class, we further extend the symbol table to method level

σ_{global}=?

$\sigma_{N.foo.start}$=?

σ_{9}=?

$\sigma_{N.foo.stop}$=?

Hash Table Implementation

- Hash table
  - Operations: hash(k), insert (k, v), lookup (k), delete(k)
  - The keyword k is often the variable name, the v is often the type of the variable (which could be a primitive type or a pointer)
  - The benefits: quick look up, easy extension from an existing symbol table to a new symbol table and easy recovery.
- The hash table representations of the previous σ

Constructing Symbol Tables

Stmt ::= Stmt; Stmt |
DeclStmt |
AssignStmt |
ReturnStmt | …

DeclStmt :: = ***int id*** | ***string id***

AssignStmt ::= ***id =*** Exp

ReturnStmt ::= ***return***

Exp ::= …

```
Stack S;
public void visit(IntDeclStmt s) {
    σ.insert(s.id, INT);
    S.add(s.id);
 }
 public void visit(StringDelStmt s) {
    σ.insert(s.id, STRING);
    S.add(s.id);
}
public  void visit(ReturnStmt s) {
    while (S.top()!='$') {
        σ.removeFirstOne(S.pop());
    }
}
public void visit(CompoundStmt s) {
     s.s1.accept();
     s.s2.accept();
}
public void visit(FunEntry s) {
     S.push('$');
```

An Example

For example, see how we update the symbol table for function foo() according to the previous defined visitor

```
int a;
int foo () {
   int b;
   a=10;
   string a;
   a=10;
   return;
}
```

Type Checking

The type checking process can be implemented through a visitor. Assume σ always represents the current symbol table.
The key is that we produce a type for EACH AST node during the traversal.

Stmt ::= Stmt; Stmt |
DeclStmt |
AssignStmt |
IfStmt | …

DeclStmt :: = ***int id*** | ***string id***

AssignStmt ::= ***id =*** Exp

IfStmt ::= ***if*** (Exp) ***{*** Stmt ***}***

Exp ::= Exp + Exp |
Exp – Exp |
id |
num |
char* | …

```
public Type visit(CompositeStmt s)  {
    s.s1.accept(this);
    s.s2.accept(this);
    return void;
}
public Type visit(StringDelStmt s) {
    return void;}
public  Type visit(AssignStmt s) {
    Type t=s.s1.accept(this);
    if (t != σ.lookup (s.id)) typeError();
    return t;
}
public  Type visit (PlusExpr e) {
    Type t1=e.e1.accept(this);
    Type t2=e.e2.accept(this);
    if (t1==t2==INT || t1==t2==STRING)
       return t1
     else TypeError();
}
```