The limitations of CFG

• Given the following grammar

\[
\begin{align*}
S & ::= \text{Decl} \mid \text{Stmt} \\
\text{Decl} & ::= \text{Type} \mid \text{id} \mid \text{Decl} \mid \text{Decl} \\
\text{Type} & ::= \text{string} \mid \text{int} \\
\text{Stmt} & ::= \text{Stmt} \mid \text{Stmt} \mid \text{id} = \text{Exp} \mid \ldots \\
\text{Exp} & ::= \text{Exp} * \text{Exp} \mid \text{id} \mid \text{num} \mid \text{char}^* \mid \ldots
\end{align*}
\]

• Does the corresponding parser accept the following programs?

```
string x;
int z;
x = “hello world”;
z = x + 1;
```
```
int x;
int z;
x = 0;
z = x + 1;
```
```
int x;
```
```
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

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.

\[
\sigma_{\text{global}} = ? \\
\sigma_{\text{N.foo.start}} = ? \\
\sigma_{9} = ? \\
\sigma_{\text{N.foo.stop}} = ?
\]
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

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()!='$') {
        σ.removeFirstOne(S.pop());
    }
}
public void visit(CompoundStmt s) {
    s.s1.accept();
    s.s2.accept();
}
public void visit(FunEntry s) {
    S.push('$');
For example, see how we update the symbol table for function foo() according to the previous defined visitor

```c
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 $\sigma$ 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 != $\sigma$.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();
}