Question: Can I implement a compiler by inserting code into the parser

- If you are asked to write a robot controller, which is indeed a compiler that interprets commands (with parameters) from the console.
  - Yes, very likely (because commands are independent)
- If you are asked to write a general program language compiler
  - No, very likely insufficient, or at least highly inefficient.

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,…)
  - When a variable is defined at S1 and then used at S2. We have to make sure the same memory location is assigned to the variable and the use sees the value of the definition
  - …

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

```
class E {
    static int a = 5;
}
class N {
    static int b = 10;
    static int a = E.a + D.d;
    public int start (int p, int bb) {
       int a;
       …
     }
     public boolean stop (int p) {
       return false;
     }
}
class D {
    static int d = E.a + N.a;
     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}=?

σ_{N}=?

$\sigma_{N.start}$=?

$\sigma_{N.stop}$=?

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 class visit(IntDeclStmt s)  {
    σ.insert(s.id, INT);
    S.add(s.id);
 }
 public class visit(StringDelStmt s) {
    σ.insert(s.id, STRING);
    S.add(s.id);
}
public  class visit(ReturnStmt s) {
    while (!S.empty()) {
        σ.pop(S.pop());
    }
}
```

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();
}
```