Computer Languages

Type checking

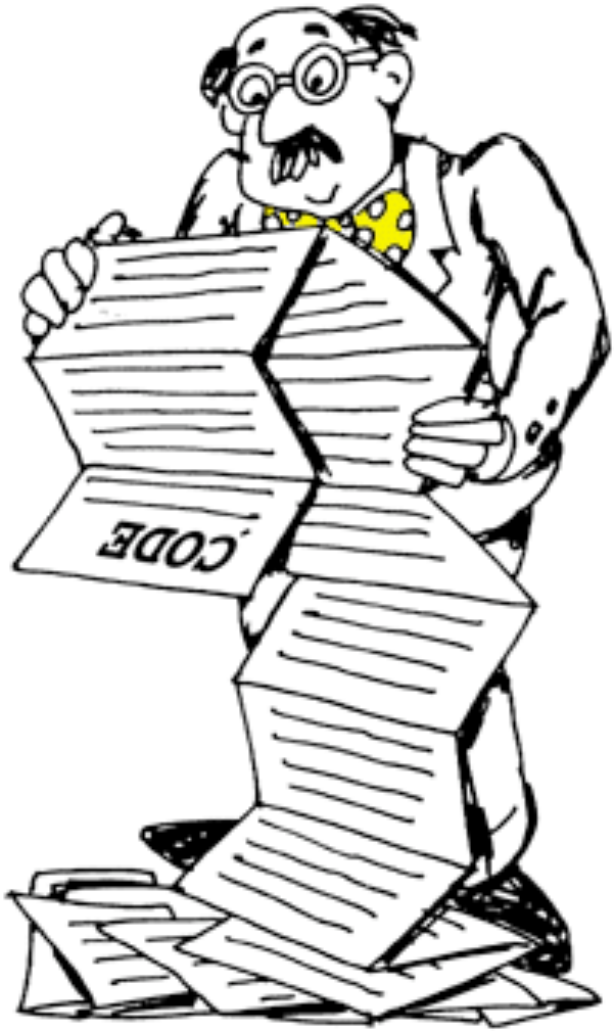
Verónica Gaspes

School of Information Science, Computer and Electrical Engineering

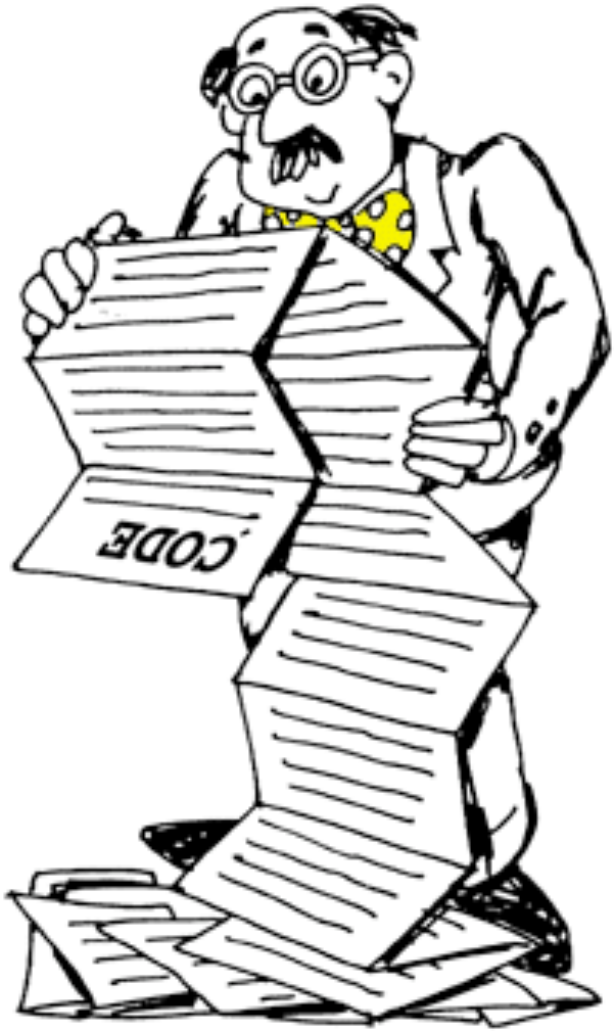
February 18

www2.hh.se/staff/vero/languages

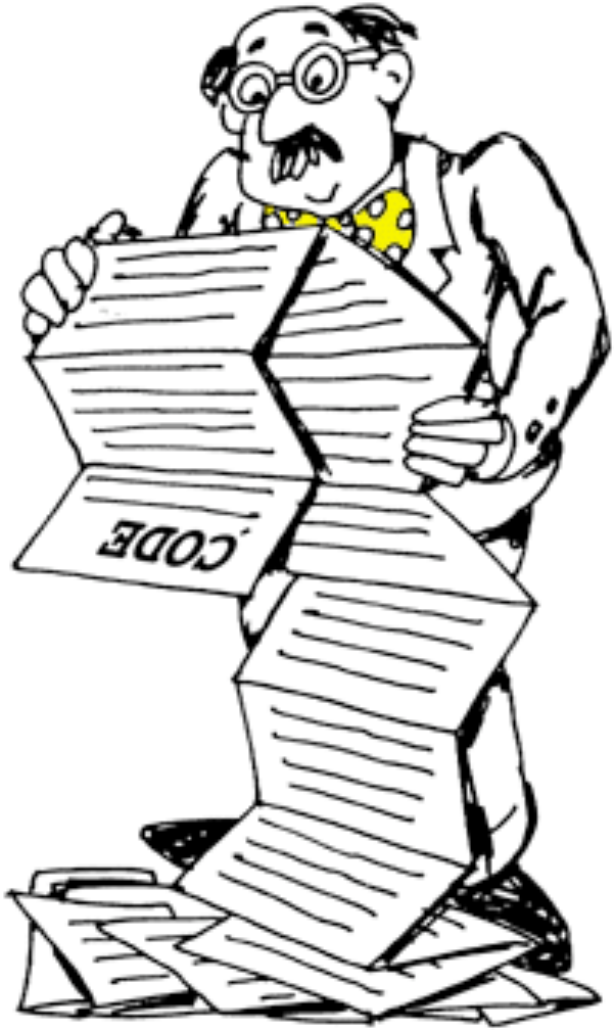
Overview of a compiler



Overview of a compiler



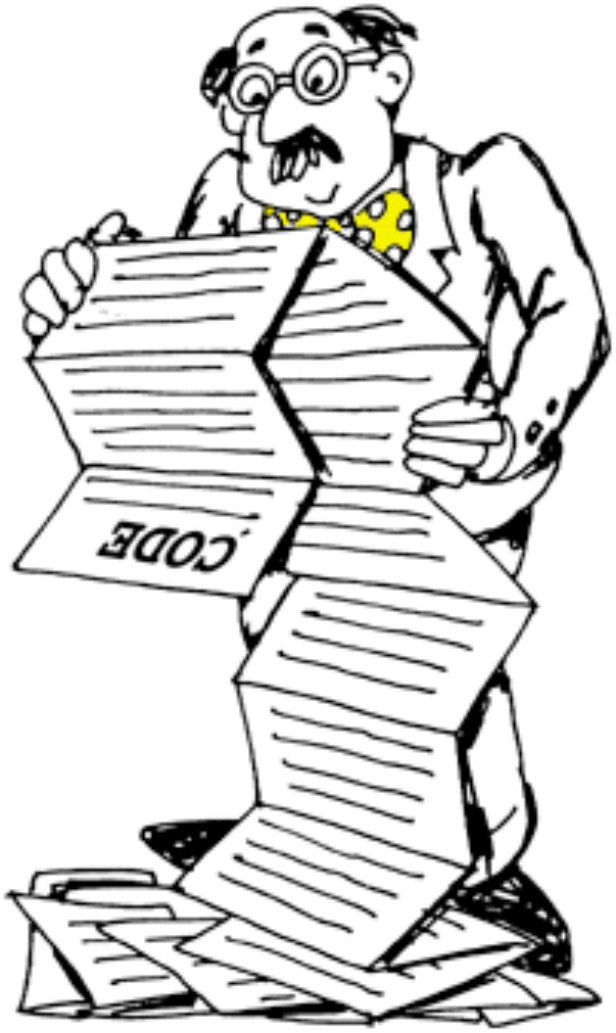
Overview of a compiler



The compiler has to

- Analyze the source code to understand what it means!
  - in the process of doing so it has to reject meaningless sources!
- Generate code in a language for which a machine exists.

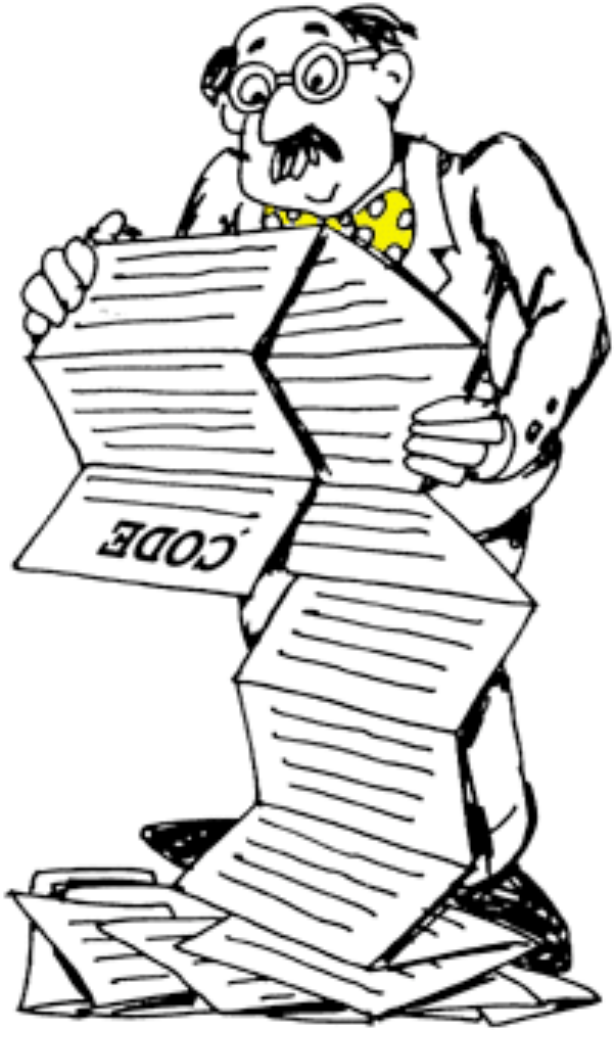
Overview of a compiler



The compiler has to

- Analyze the source code to understand what it means!
  - in the process of doing so it has to reject meaningless sources!
- Generate code in a language for which a machine exists.

Overview of a compiler

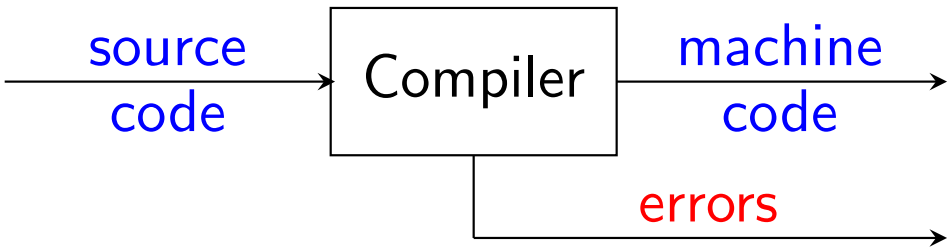


The compiler has to

- Analyze the source code to understand what it means!
  - in the process of doing so it has to reject meaningless sources!
- Generate code in a language for which a machine exists.

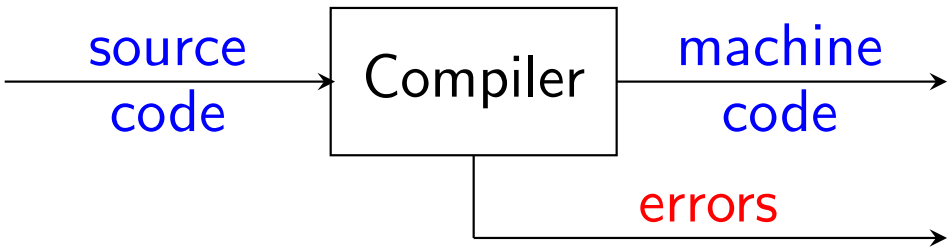
Overview of a compiler

Overview of a compiler



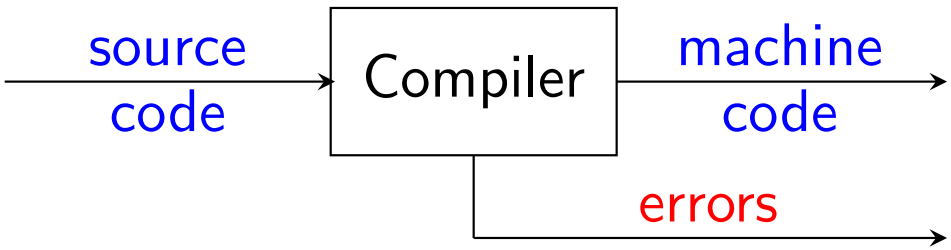
- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!

Overview of a compiler



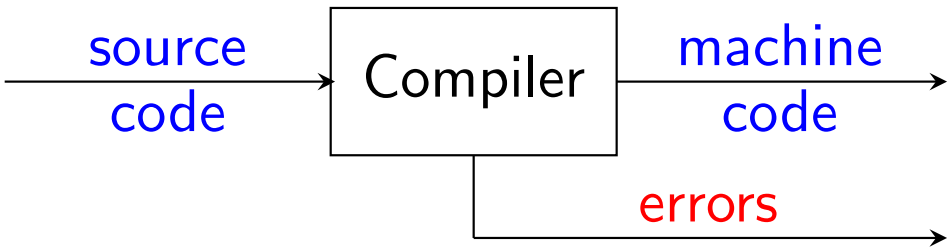
- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!

Overview of a compiler



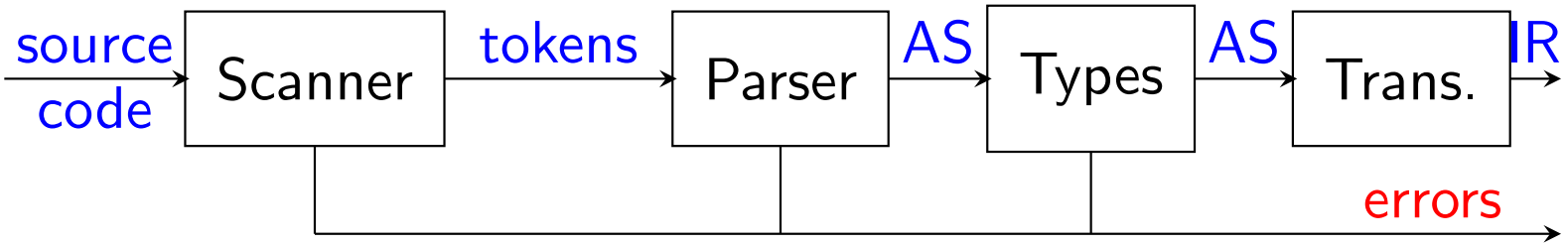
- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!

Overview of a compiler



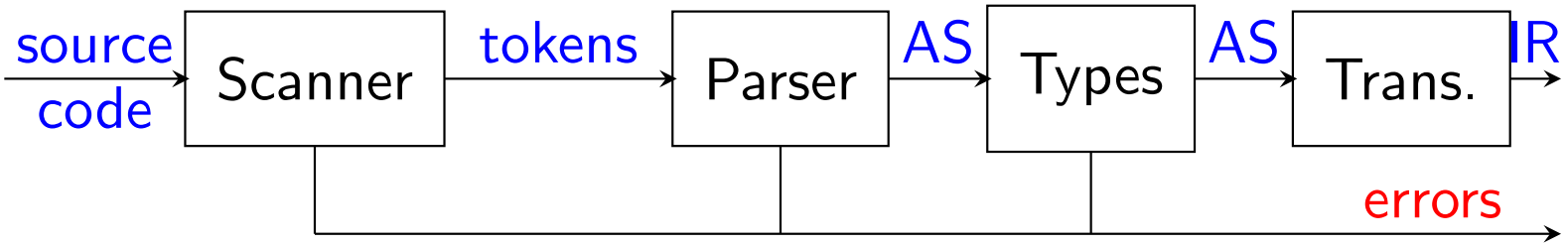
- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!

The Front End (analysis phase)



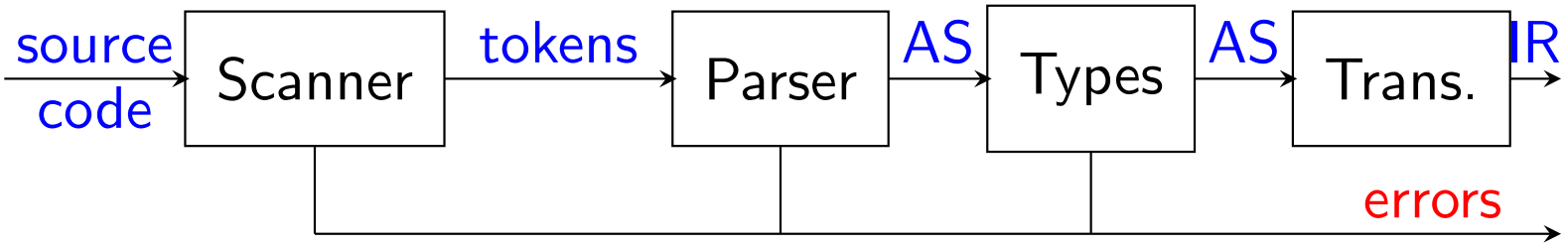
- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed by the type checker and is then used to generate the intermediate representation.

The Front End (analysis phase)



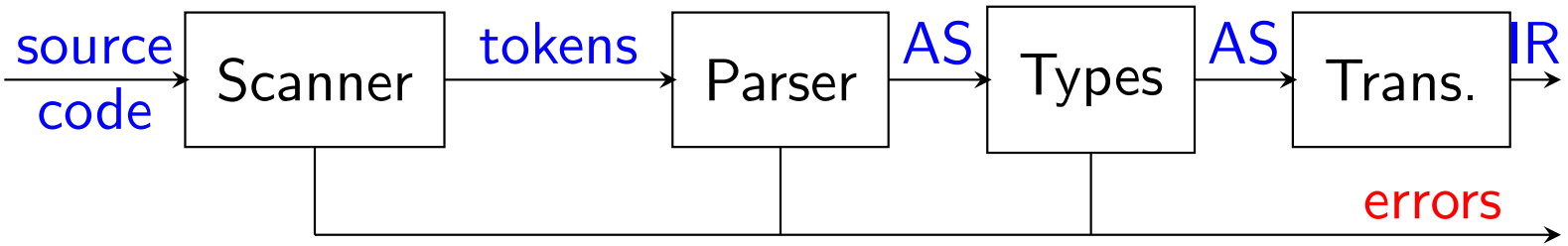
- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed by the type checker and is then used to generate the intermediate representation.

The Front End (analysis phase)



- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed by the type checker and is then used to generate the intermediate representation.

The Front End (analysis phase)



- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed by the type checker and is then used to generate the intermediate representation.

Environments

- The parser has produced an abstract syntax tree representation of the source program.
- The typechecker will inspect this representation to see whether it is meaningful.

Inspecting declarations

- Bind identifiers with usefull information for look up when inspecting uses of identifiers. *Build an environment!*
- Make required controls as specified by the type system.

Inspecting statements and expressions

Make required controls as specified by the type system. The types of the sub*expressions* are required!

Environments

- The parser has produced an abstract syntax tree representation of the source program.
- The typechecker will inspect this representation to see whether it is meaningful.

Inspecting declarations

- Bind identifiers with usefull information for look up when inspecting uses of identifiers. *Build an environment!*
- Make required controls as specified by the type system.

Inspecting statements and expressions

Make required controls as specified by the type system. The types of the sub*expressions* are required!

Environments

- The parser has produced an abstract syntax tree representation of the source program.
- The typechecker will inspect this representation to see whether it is meaningful.

Inspecting declarations

- Bind identifiers with usefull information for look up when inspecting uses of identifiers. *Build an environment!*
- Make required controls as specified by the type system.

Inspecting statements and expressions

Make required controls as specified by the type system. The types of the sub*expressions* are required!

Environments

- The parser has produced an abstract syntax tree representation of the source program.
- The typechecker will inspect this representation to see whether it is meaningful.

Inspecting declarations

- Bind identifiers with usefull information for look up when inspecting uses of identifiers. *Build an environment!*
- Make required controls as specified by the type system.

Inspecting statements and expressions

Make required controls as specified by the type system. The types of the sub*expressions* are required!

Environments

- The parser has produced an abstract syntax tree representation of the source program.
- The typechecker will inspect this representation to see whether it is meaningful.

Inspecting declarations

- Bind identifiers with usefull information for look up when inspecting uses of identifiers. *Build an environment!*
- Make required controls as specified by the type system.

Inspecting statements and expressions

Make required controls as specified by the type system. The types of the sub*expressions* are required!

Environments

- The parser has produced an abstract syntax tree representation of the source program.
- The typechecker will inspect this representation to see whether it is meaningful.

Inspecting declarations

- Bind identifiers with usefull information for look up when inspecting uses of identifiers. *Build an environment!*
- Make required controls as specified by the type system.

Inspecting statements and expressions

Make required controls as specified by the type system. The types of the sub*expressions* are required!

The global scope

All classes and their methods form a global scope for the program.

- There cannot be classes with the same name!
- In each class there cannot be methods with the same name and the same *signature*

```
class Factorial{
   public static void main(String[] a){
      System.out.println(new Fac().ComputeFac(10));
   }
}
class Fac {
   public int ComputeFac(int num){
      int num_aux ;...
      return num_aux ;
   }
}
```

The global scope

All classes and their methods form a global scope for the program.

- There cannot be classes with the same name!
- In each class there cannot be methods with the same name and the same *signature*

```
class Factorial{
   public static void main(String[] a){
      System.out.println(new Fac().ComputeFac(10));
   }
}
class Fac {
   public int ComputeFac(int num){
      int num_aux ;...
      return num_aux ;
   }
}
```

The global scope

All classes and their methods form a global scope for the program.

- There cannot be classes with the same name!
- In each class there cannot be methods with the same name and the same *signature*

```
class Factorial{
   public static void main(String[] a){
      System.out.println(new Fac().ComputeFac(10));
   }
}
class Fac {
   public int ComputeFac(int num){
      int num_aux ;...
      return num_aux ;
   }
}
```

The *types* of classes and methods

What do we have to bind to each class identifier in this global scope?

Its interface: what methods it has!

What do we have to bind to each method identifier in the class interface?

Its signature: the type of the result and the type of the arguments in order.

The *types* of classes and methods

What do we have to bind to each class identifier in this global scope?

Its interface: what methods it has!

What do we have to bind to each method identifier in the class interface?

Its signature: the type of the result and the type of the arguments in order.

The *types* of classes and methods

What do we have to bind to each class identifier in this global scope?

Its interface: what methods it has!

What do we have to bind to each method identifier in the class interface?

Its signature: the type of the result and the type of the arguments in order.

The *types* of classes and methods

What do we have to bind to each class identifier in this global scope?

Its interface: what methods it has!

What do we have to bind to each method identifier in the class interface?

Its signature: the type of the result and the type of the arguments in order.

The *types* of classes and methods

What do we have to bind to each class identifier in this global scope?

Its interface: what methods it has!

What do we have to bind to each method identifier in the class interface?

Its signature: the type of the result and the type of the arguments in order.

The *types* of classes and methods

What do we have to bind to each class identifier in this global scope?

Its interface: what methods it has!

What do we have to bind to each method identifier in the class interface?

Its signature: the type of the result and the type of the arguments in order.

The *types* of classes and methods

What do we have to bind to each class identifier in this global scope?

Its interface: what methods it has!

What do we have to bind to each method identifier in the class interface?

Its signature: the type of the result and the type of the arguments in order.

Implementing interfaces and signatures

```
class Interface {
    // the "type" of a class in minijava
    Map<String,Signature> methods;
    String parent;
}
class Signature {
    // the "type" of methods in minijava
    Type returnType;
    List<Type> formalTypes;
}
```

These can be calculated by visiting a program, visiting the class declarations and their method declarations (without going deeper!)

The only thing that has to be checked is that no class names are repeated and that no methods are repeated inside a class.

Implementing interfaces and signatures

```
class Interface {
    // the "type" of a class in minijava
    Map<String,Signature> methods;
    String parent;
}
class Signature {
    // the "type" of methods in minijava
    Type returnType;
    List<Type> formalTypes;
}
```

These can be calculated by visiting a program, visiting the class declarations and their method declarations (without going deeper!)

The only thing that has to be checked is that no class names are repeated and that no methods are repeated inside a class.

Implementing interfaces and signatures

```
class Interface {
    // the "type" of a class in minijava
    Map<String,Signature> methods;
    String parent;
}
class Signature {
    // the "type" of methods in minijava
    Type returnType;
    List<Type> formalTypes;
}
```

These can be calculated by visiting a program, visiting the class declarations and their method declarations (without going deeper!)

The only thing that has to be checked is that no class names are repeated and that no methods are repeated inside a class.

The TypeChecker

This is how the TypeChecker starts . . .

```
public static void typeCheck(Program p){
    SymbolTable env = new SymbolTable();
    env = p.accept(new DecElaborator(),env);
          ...
}
```

The DecElaborator

This is what the DecElaborator does when visiting a program

```
class DecElaborator
implements Visitor<SymbolTable, SymbolTable>{


    public SymbolTable visit(Program n, SymbolTable env){
        env = n.m.accept(this,env);
        for(ClassDecl c : n.cl){
            env = c.accept(this,env);
        }
        return env;
    }
```

The DecElaborator

This is what the DecElaborator does when visiting a class declaration

```
public SymbolTable visit(ClassDeclExtends n, SymbolTable e
    Interface _interface = new Interface();
    _interface.parent = n.j.s;
    _interface.fields = n.vl;
    for(MethodDecl m : n.ml){
        _interface.addMethod(m.i.s, new Signature(m));
    }
    env.addClass(n.i.s, _interface);
    return env;
}
```

No more *accept* calls!

The SymbolTable

Part of the symbol table will record this global scope and provide the methods to update it

```
class SymbolTable {
Map<String,Interface> classes =
    new HashMap<String,Interface>();

public void addClass(String className,
                     Interface classInterface){
    if(classes.containsKey(className)){
        System.out.println("<< "+className+ " >>
                                already defined");
        System.exit(1);
    }
    classes.put(className, classInterface);
}
```

The rest of type checking

When the global scope has been recorded in the symbol table, each method in each class has to be type checked.

It has to be checked that the rules from the type system for minijava are enforced.

```
public static void typeCheck(Program p){
    SymbolTable env = new SymbolTable();

    env = p.accept(new DecElaborator(),env);
    p.accept(new TypeChecker(),env);
}
```

The rest of the type checker

For each class:

1. Start a new local scope and add all the fields of this class
2. Typecheck all the methods in the new environment

Looking ahead: What is the type of `this`?

The rest of the type checker

For each class:

1. Start a new local scope and add all the fields of this class
2. Typecheck all the methods in the new environment

Looking ahead: What is the type of `this`?

The rest of the type checker

For each class:

1. Start a new local scope and add all the fields of this class
2. Typecheck all the methods in the new environment

Looking ahead: What is the type of `this`?

The rest of the type checker

For each class:

1. Start a new local scope and add all the fields of this class
2. Typecheck all the methods in the new environment

Looking ahead: What is the type of `this`?

The local scopes

When classes and methods have to be typed checked, local scopes with fields, parameters and local variables have to be temporarilly introduced.

When visiting a class, a local scope representing the fields has to be added to the identifiers that can be used in the scope.

```
List<Map<String,Type>> localScopes
```

All this information has to be built up and stored in a datastructure so that when the identifiers are used they can be looked up!

The local scopes

When classes and methods have to be typed checked, local scopes with fields, parameters and local variables have to be temporarilly introduced.

When visiting a class, a local scope representing the fields has to be added to the identifiers that can be used in the scope.

```
List<Map<String,Type>> localScopes
```

All this information has to be built up and stored in a datastructure so that when the identifiers are used they can be looked up!

The local scopes

When classes and methods have to be typed checked, local scopes with fields, parameters and local variables have to be temporarilly introduced.

When visiting a class, a local scope representing the fields has to be added to the identifiers that can be used in the scope.

```
List<Map<String,Type>> localScopes
```

All this information has to be built up and stored in a datastructure so that when the identifiers are used they can be looked up!

The local scopes

When classes and methods have to be typed checked, local scopes with fields, parameters and local variables have to be temporarilly introduced.

When visiting a class, a local scope representing the fields has to be added to the identifiers that can be used in the scope.

```
List<Map<String,Type>> localScopes
```

All this information has to be built up and stored in a datastructure so that when the identifiers are used they can be looked up!

The local scopes

When classes and methods have to be typed checked, local scopes with fields, parameters and local variables have to be temporarilly introduced.

When visiting a class, a local scope representing the fields has to be added to the identifiers that can be used in the scope.

```
List<Map<String,Type>> localScopes
```

All this information has to be built up and stored in a datastructure so that when the identifiers are used they can be looked up!

The symbol table

```
class SymbolTable {
  Map<String,Interface> classes =
     new HashMap<String,Interface>();
  List<Map<String,Type>> localScopes =
     new LinkedList<Map<String,Type>>();
  public void addClass(String className,
                       Interface classInterface){
     ...
  }
  public void removeLocalScope(){
    localScopes.removeLast();
  }
  ...
}
```

The symbol table

```
class SymbolTable {
  Map<String,Interface> classes =
     new HashMap<String,Interface>();
  List<Map<String,Type>> localScopes =
     new LinkedList<Map<String,Type>>();
  public void addClass(String className,
                       Interface classInterface){
     ...
  }
  public void removeLocalScope(){
    localScopes.removeLast();
  }
  ...
}
```

The symbol table

```
class SymbolTable {
  Map<String,Interface> classes =
     new HashMap<String,Interface>();
  List<Map<String,Type>> localScopes =
     new LinkedList<Map<String,Type>>();
  public void addClass(String className,
                       Interface classInterface){
     ...
  }
  public void removeLocalScope(){
    localScopes.removeLast();
  }
  ...
}
```

The symbol table

```
class SymbolTable {
  Map<String,Interface> classes =
     new HashMap<String,Interface>();
  List<Map<String,Type>> localScopes =
     new LinkedList<Map<String,Type>>();
  public void addClass(String className,
                       Interface classInterface){
     ...
  }
  public void removeLocalScope(){
    localScopes.removeLast();
  }
  ...
}
```

The symbol table

```
class SymbolTable {
  Map<String,Interface> classes =
     new HashMap<String,Interface>();
  List<Map<String,Type>> localScopes =
     new LinkedList<Map<String,Type>>();
  public void addClass(String className,
                       Interface classInterface){
     ...
  }
  public void removeLocalScope(){
    localScopes.removeLast();
  }
  ...
}
```

The symbol table

```
class SymbolTable {
  Map<String,Interface> classes =
     new HashMap<String,Interface>();
  List<Map<String,Type>> localScopes =
     new LinkedList<Map<String,Type>>();
  public void addClass(String className,
                       Interface classInterface){
     ...
  }
  public void removeLocalScope(){
    localScopes.removeLast();
  }
  ...
}
```

Typechecking methods

What remains to be done on each method

1. Start a new local scope for the arguments and local variables
2. Typecheck the statements in the body of the method
3. Check that the type of the expresion in the return is a subtype of the return type of the method!

Typechecking methods

What remains to be done on each method

1. Start a new local scope for the arguments and local variables
2. Typecheck the statements in the body of the method
3. Check that the type of the expresion in the return is a subtype of the return type of the method!

Typechecking methods

What remains to be done on each method

1. Start a new local scope for the arguments and local variables
2. Typecheck the statements in the body of the method
3. Check that the type of the expresion in the return is a subtype of the return type of the method!

Typechecking methods

What remains to be done on each method

1. Start a new local scope for the arguments and local variables
2. Typecheck the statements in the body of the method
3. Check that the type of the expresion in the return is a subtype of the return type of the method!

Typechecking statements and expressions

```
class TypeChecker implements Visitor<Type, SymbolTable>{
  public Type visit(Program n, SymbolTable env){
    n.m.accept(this,env);
    for(ClassDecl c : n.cl)
      c.accept(this,env);
    return null;
  }
  public Type visit(Plus n, SymbolTable env){...}
  ...
}
```

Typechecking statements and expressions

```
class TypeChecker implements Visitor<Type, SymbolTable>{
  public Type visit(Program n, SymbolTable env){
    n.m.accept(this,env);
    for(ClassDecl c : n.cl)
      c.accept(this,env);
    return null;
  }
  public Type visit(Plus n, SymbolTable env){...}
  ...
}
```

Typechecking statements and expressions

```
class TypeChecker implements Visitor<Type, SymbolTable>{
  public Type visit(Program n, SymbolTable env){
    n.m.accept(this,env);
    for(ClassDecl c : n.cl)
      c.accept(this,env);
    return null;
  }
  public Type visit(Plus n, SymbolTable env){...}
  ...
}
```

Typechecking statements and expressions

```
class TypeChecker implements Visitor<Type, SymbolTable>{
  public Type visit(Program n, SymbolTable env){
    n.m.accept(this,env);
    for(ClassDecl c : n.cl)
      c.accept(this,env);
    return null;
  }
  public Type visit(Plus n, SymbolTable env){...}
  ...
}
```

Errors

Where will errors be detected?

Example

- When trying to add class to the global scope and there is already a class with that name!
- When trying to add a method to an interface and there is already a method with the same name and signature there!
- When adding a variable to a local context and there is already a variable with the same name in the same local context!
- When looking up an identifier and finding that it is not in scope!
- When comparing expected types with calculated types!
- . . . ?

Errors

Where will errors be detected?

Example

- When trying to add class to the global scope and there is already a class with that name!
- When trying to add a method to an interface and there is already a method with the same name and signature there!
- When adding a variable to a local context and there is already a variable with the same name in the same local context!
- When looking up an identifier and finding that it is not in scope!
- When comparing expected types with calculated types!
- . . . ?

Errors

Where will errors be detected?

Example

- When trying to add class to the global scope and there is already a class with that name!
- When trying to add a method to an interface and there is already a method with the same name and signature there!
- When adding a variable to a local context and there is already a variable with the same name in the same local context!
- When looking up an identifier and finding that it is not in scope!
- When comparing expected types with calculated types!
- . . . ?

Errors

Where will errors be detected?

Example

- When trying to add class to the global scope and there is already a class with that name!
- When trying to add a method to an interface and there is already a method with the same name and signature there!
- When adding a variable to a local context and there is already a variable with the same name in the same local context!
- When looking up an identifier and finding that it is not in scope!
- When comparing expected types with calculated types!
- . . . ?

Errors

Where will errors be detected?

Example

- When trying to add class to the global scope and there is already a class with that name!
- When trying to add a method to an interface and there is already a method with the same name and signature there!
- When adding a variable to a local context and there is already a variable with the same name in the same local context!
- When looking up an identifier and finding that it is not in scope!
- When comparing expected types with calculated types!
- . . . ?

Errors

Where will errors be detected?

Example

- When trying to add class to the global scope and there is already a class with that name!
- When trying to add a method to an interface and there is already a method with the same name and signature there!
- When adding a variable to a local context and there is already a variable with the same name in the same local context!
- When looking up an identifier and finding that it is not in scope!
- When comparing expected types with calculated types!
- . . . ?

Errors

Where will errors be detected?

Example

- When trying to add class to the global scope and there is already a class with that name!
- When trying to add a method to an interface and there is already a method with the same name and signature there!
- When adding a variable to a local context and there is already a variable with the same name in the same local context!
- When looking up an identifier and finding that it is not in scope!
- When comparing expected types with calculated types!
- . . . ?

Errors

Where will errors be detected?

Example

- When trying to add class to the global scope and there is already a class with that name!
- When trying to add a method to an interface and there is already a method with the same name and signature there!
- When adding a variable to a local context and there is already a variable with the same name in the same local context!
- When looking up an identifier and finding that it is not in scope!
- When comparing expected types with calculated types!
- . . . ?