Computer Languages

Static semantics – contextual analysis

February 11

Static Semantics (or Contextual Analysis)

- The syntax of computer languages is described using *context free grammars* because it is easy to construct parsers for them!
- Context free grammars cannot be used to describe context constraints!
  - Scope rules
  - Type rules
  - Flow of control checks
  - Uniqueness checks
- These analysis are done
  - *On* the abstract syntax tree produced by the parser.
  - *Using* environments constructed by elaborating declarations.

Static Semantics (or Contextual Analysis)

- The syntax of computer languages is described using *context free grammars* because it is easy to construct parsers for them!
- Context free grammars cannot be used to describe

context constraints!

- Scope constraints
- Type constraints

- These analysis are done
  - *On* the abstract syntax tree produced by the parser.
  - *Using* environments constructed by elaborating declarations.

Static Semantics (or Contextual Analysis)

- The syntax of computer languages is described using *context free grammars* because it is easy to construct parsers for them!
- Context free grammars cannot be used to describe

context constraints!

- Scope constraints
  - *Where* can identifiers be used according to where they are declared?
- Type constraints
  - *How* can identifiers be used according to their declaration?
  - *What* expressions are meaningful?

- These analysis are done
  - *On* the abstract syntax tree produced by the parser.
  - *Using* environments constructed by elaborating declarations.

Static Semantics (or Contextual Analysis)

- The syntax of computer languages is described using *context free grammars* because it is easy to construct parsers for them!
- Context free grammars cannot be used to describe

context constraints!

- Scope constraints
  - *Where* can identifiers be used according to where they are declared?
- Type constraints
  - *How* can identifiers be used according to their declaration?
  - *What* expressions are meaningful?

- These analysis are done
  - *On* the abstract syntax tree produced by the parser.
  - *Using* environments constructed by elaborating declarations.

Static Semantics (or Contextual Analysis)

- The syntax of computer languages is described using *context free grammars* because it is easy to construct parsers for them!
- Context free grammars cannot be used to describe

context constraints!

- Scope constraints
  - *Where* can identifiers be used according to where they are declared?
- Type constraints
  - *How* can identifiers be used according to their declaration?
  - *What* expressions are meaningful?

- These analysis are done
  - *On* the abstract syntax tree produced by the parser.
  - *Using* environments constructed by elaborating declarations.

Static Semantics (or Contextual Analysis)

- The syntax of computer languages is described using *context free grammars* because it is easy to construct parsers for them!
- Context free grammars cannot be used to describe

context constraints!

- Scope constraints
  - *Where* can identifiers be used according to where they are declared?
- Type constraints
  - *How* can identifiers be used according to their declaration?
  - *What* expressions are meaningful?

- These analysis are done
  - *On* the abstract syntax tree produced by the parser.
  - *Using* environments constructed by elaborating declarations.

Static Semantics (or Contextual Analysis)

- The syntax of computer languages is described using *context free grammars* because it is easy to construct parsers for them!
- Context free grammars cannot be used to describe

context constraints!

- Scope constraints
  - *Where* can identifiers be used according to where they are declared?
- Type constraints
  - *How* can identifiers be used according to their declaration?
  - *What* expressions are meaningful?

- These analysis are done
  - *On* the abstract syntax tree produced by the parser.
  - *Using* environments constructed by elaborating declarations.

Static Semantics (or Contextual Analysis)

- The syntax of computer languages is described using *context free grammars* because it is easy to construct parsers for them!
- Context free grammars cannot be used to describe

context constraints!

- Scope constraints
  - *Where* can identifiers be used according to where they are declared?
- Type constraints
  - *How* can identifiers be used according to their declaration?
  - *What* expressions are meaningful?

- These analysis are done
  - *On* the abstract syntax tree produced by the parser.
  - *Using* environments constructed by elaborating declarations.

Static Semantics (or Contextual Analysis)

- The syntax of computer languages is described using *context free grammars* because it is easy to construct parsers for them!
- Context free grammars cannot be used to describe

context constraints!

- Scope constraints
  - *Where* can identifiers be used according to where they are declared?
- Type constraints
  - *How* can identifiers be used according to their declaration?
  - *What* expressions are meaningful?

- These analysis are done
  - *On* the abstract syntax tree produced by the parser.
  - *Using* environments constructed by elaborating declarations.

Static Semantics (or Contextual Analysis)

- The syntax of computer languages is described using *context free grammars* because it is easy to construct parsers for them!
- Context free grammars cannot be used to describe

context constraints!

- Scope constraints
  - *Where* can identifiers be used according to where they are declared?
- Type constraints
  - *How* can identifiers be used according to their declaration?
  - *What* expressions are meaningful?

- These analysis are done
  - *On* the abstract syntax tree produced by the parser.
  - *Using* environments constructed by elaborating declarations.

Static Semantics (or Contextual Analysis)

- The syntax of computer languages is described using *context free grammars* because it is easy to construct parsers for them!
- Context free grammars cannot be used to describe

context constraints!

- Scope constraints
  - *Where* can identifiers be used according to where they are declared?
- Type constraints
  - *How* can identifiers be used according to their declaration?
  - *What* expressions are meaningful?

- These analysis are done
  - *On* the abstract syntax tree produced by the parser.
  - *Using* environments constructed by elaborating declarations.

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }

}
```

A class opens a scope adding names for fields and methods

A method opens a scope adding names for arguments and variables

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }
}
```

A class opens a scope adding names for fields and methods

A method opens a scope adding names for arguments and variables

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }
}
```

A class opens a scope adding names for fields and methods

A method opens a scope adding names for arguments and variables

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }
}
```

A class opens a scope adding names for fields and methods

A method opens a scope adding names for arguments and variables

Scope analysis

- The compiler has to generate code for some target machine.
- Part of this involves generating code to deal with the identifiers used in the program.
- The compiler has to understand what every identifier stands for!

Elaborating declarations

- Declarations introduce identifiers and indicate what they stand for.
- The compiler has to record this in a datastructure for looking up when it finds a use of the identifer!

Some languages require definitions before use, others don't (like `minijava`). In the latter case, semantic analysis requires 2 phases!

Scope analysis

- The compiler has to generate code for some target machine.
- Part of this involves generating code to deal with the identifiers used in the program.
- The compiler has to understand what every identifier stands for!

Elaborating declarations

- Declarations introduce identifiers and indicate what they stand for.
- The compiler has to record this in a datastructure for looking up when it finds a use of the identifer!

Some languages require definitions before use, others don't (like `minijava`). In the latter case, semantic analysis requires 2 phases!

Scope analysis

- The compiler has to generate code for some target machine.
- Part of this involves generating code to deal with the identifiers used in the program.
- The compiler has to understand what every identifier stands for!

Elaborating declarations

- Declarations introduce identifiers and indicate what they stand for.
- The compiler has to record this in a datastructure for looking up when it finds a use of the identifer!

Some languages require definitions before use, others don't (like `minijava`). In the latter case, semantic analysis requires 2 phases!

Scope analysis

- The compiler has to generate code for some target machine.
- Part of this involves generating code to deal with the identifiers used in the program.
- The compiler has to understand what every identifier stands for!

Elaborating declarations

- Declarations introduce identifiers and indicate what they stand for.
- The compiler has to record this in a datastructure for looking up when it finds a use of the identifer!

Some languages require definitions before use, others don't (like `minijava`). In the latter case, semantic analysis requires 2 phases!

Scope analysis

- The compiler has to generate code for some target machine.
- Part of this involves generating code to deal with the identifiers used in the program.
- The compiler has to understand what every identifier stands for!

Elaborating declarations

- Declarations introduce identifiers and indicate what they stand for.
- The compiler has to record this in a datastructure for looking up when it finds a use of the identifer!

Some languages require definitions before use, others don't (like `minijava`). In the latter case, semantic analysis requires 2 phases!

Scope analysis

- The compiler has to generate code for some target machine.
- Part of this involves generating code to deal with the identifiers used in the program.
- The compiler has to understand what every identifier stands for!

Elaborating declarations

- Declarations introduce identifiers and indicate what they stand for.
- The compiler has to record this in a datastructure for looking up when it finds a use of the identifer!

Some languages require definitions before use, others don't (like `minijava`). In the latter case, semantic analysis requires 2 phases!

Scope analysis

- The compiler has to generate code for some target machine.
- Part of this involves generating code to deal with the identifiers used in the program.
- The compiler has to understand what every identifier stands for!

Elaborating declarations

- Declarations introduce identifiers and indicate what they stand for.
- The compiler has to record this in a datastructure for looking up when it finds a use of the identifer!

Some languages require definitions before use, others don't (like `minijava`). In the latter case, semantic analysis requires 2 phases!

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }

}
```

- All methods in the class can use the fields `l` and `r` as of type `Tree`.
- Inside the method `visit` also the argument `n` and the variable `nti`
- We expect the class `Tree` to include methods

```
boolean GetHas_Right()

boolean GetHas_Left()

Tree GetRight()

Tree GetLeft()

int accept(Visitor)
```

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }

}
```

- All methods in the class can use the fields `l` and `r` as of type `Tree`.
- Inside the method `visit` also the argument `n` and the variable `nti`
- We expect the class `Tree` to include methods

```
boolean GetHas_Right()

boolean GetHas_Left()

Tree GetRight()

Tree GetLeft()

int accept(Visitor)
```

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }

}
```

- All methods in the class can use the fields `l` and `r` as of type `Tree`.
- Inside the method `visit` also the argument `n` and the variable `nti`
- We expect the class `Tree` to include methods

```
boolean GetHas_Right()

boolean GetHas_Left()

Tree GetRight()

Tree GetLeft()

int accept(Visitor)
```

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }

}
```

- All methods in the class can use the fields `l` and `r` as of type `Tree`.
- Inside the method `visit` also the argument `n` and the variable `nti`
- We expect the class `Tree` to include methods

```
boolean GetHas_Right()

boolean GetHas_Left()

Tree GetRight()

Tree GetLeft()

int accept(Visitor)
```

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }

}
```

- All methods in the class can use the fields `l` and `r` as of type `Tree`.
- Inside the method `visit` also the argument `n` and the variable `nti`
- We expect the class `Tree` to include methods

```
boolean GetHas_Right()

boolean GetHas_Left()

Tree GetRight()

Tree GetLeft()

int accept(Visitor)
```

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }

}
```

- All methods in the class can use the fields `l` and `r` as of type `Tree`.
- Inside the method `visit` also the argument `n` and the variable `nti`
- We expect the class `Tree` to include methods

```
boolean GetHas_Right()

boolean GetHas_Left()

Tree GetRight()

Tree GetLeft()

int accept(Visitor)
```

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }

}
```

- All methods in the class can use the fields `l` and `r` as of type `Tree`.
- Inside the method `visit` also the argument `n` and the variable `nti`
- We expect the class `Tree` to include methods

```
boolean GetHas_Right()

boolean GetHas_Left()

Tree GetRight()

Tree GetLeft()

int accept(Visitor)
```

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }

}
```

- All methods in the class can use the fields `l` and `r` as of type `Tree`.
- Inside the method `visit` also the argument `n` and the variable `nti`
- We expect the class `Tree` to include methods

```
boolean GetHas_Right()

boolean GetHas_Left()

Tree GetRight()

Tree GetLeft()

int accept(Visitor)
```

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }

}
```

- All methods in the class can use the fields `l` and `r` as of type `Tree`.
- Inside the method `visit` also the argument `n` and the variable `nti`
- We expect the class `Tree` to include methods

```
boolean GetHas_Right()

boolean GetHas_Left()

Tree GetRight()

Tree GetLeft()

int accept(Visitor)
```

Static Semantics - a *minijava* example

```
class Visitor {

    Tree l ;
    Tree r ;

    public int visit(Tree n){
        int nti ;

        if (n.GetHas_Right()){
            r = n.GetRight() ;
            nti = r.accept(this) ; }
        else nti = 0 ;

        if (n.GetHas_Left()) {
            l = n.GetLeft();
            nti = l.accept(this) ; }
        else nti = 0 ;

        return 0;
    }

}
```

- All methods in the class can use the fields `l` and `r` as of type `Tree`.
- Inside the method `visit` also the argument `n` and the variable `nti`
- We expect the class `Tree` to include methods

```
boolean GetHas_Right()

boolean GetHas_Left()

Tree GetRight()

Tree GetLeft()

int accept(Visitor)
```

Types

Almost all computer languages include a

Type System

that prescribes

- Some *primitive types*
- How *new types* can be introduced
- What the *values of each type* are
- What the *legal operations* for each type are (or what the type of each operation is)
- Type constraints for the constructs of the language

Types

Almost all computer languages include a

Type System

that prescribes

- Some *primitive types*
- How *new types* can be introduced
- What the *values of each type* are
- What the *legal operations* for each type are (or what the type of each operation is)
- Type constraints for the constructs of the language

Types

Almost all computer languages include a

Type System

that prescribes

- Some *primitive types*
- How *new types* can be introduced
- What the *values of each type* are
- What the *legal operations* for each type are (or what the type of each operation is)
- Type constraints for the constructs of the language

Types

Almost all computer languages include a

Type System

that prescribes

- Some *primitive types*
- How *new types* can be introduced
- What the *values of each type* are
- What the *legal operations* for each type are (or what the type of each operation is)
- Type constraints for the constructs of the language

Types

Almost all computer languages include a

Type System

that prescribes

- Some *primitive types*
- How *new types* can be introduced
- What the *values of each type* are
- What the *legal operations* for each type are (or what the type of each operation is)
- Type constraints for the constructs of the language

Types

Almost all computer languages include a

Type System

that prescribes

- Some *primitive types*
- How *new types* can be introduced
- What the *values of each type* are
- What the *legal operations* for each type are (or what the type of each operation is)
- Type constraints for the constructs of the language

The purpose of type systems

1. Run-time safety

The compiler should eliminate as many run-time errors as it can using type-checking techniques.

2. Increased expressiveness

By using type information a language can be made more expressive! Overloading, dynamic binding!

3. Run-time efficiency

- Information from static types can be used by the compiler to generate better code.
- Run-time checks can be avoided when the compiler knows enough.

The purpose of type systems

1. Run-time safety

The compiler should eliminate as many run-time errors as it can using type-checking techniques.

2. Increased expressiveness

By using type information a language can be made more expressive! Overloading, dynamic binding!

3. Run-time efficiency

- Inferred run-time types can be used by the code generator to produce better code.
- Run-time checks can be removed when the compiler can prove they are unnecessary.

The purpose of type systems

1. Run-time safety

The compiler should eliminate as many run-time errors as it can using type-checking techniques.

2. Increased expressiveness

By using type information a language can be made more expressive! Overloading, dynamic binding!

3. Run-time efficiency

The purpose of type systems

1. Run-time safety

The compiler should eliminate as many run-time errors as it can using type-checking techniques.

2. Increased expressiveness

By using type information a language can be made more expressive! Overloading, dynamic binding!

3. Run-time efficiency

The purpose of type systems

1. Run-time safety

The compiler should eliminate as many run-time errors as it can using type-checking techniques.

2. Increased expressiveness

By using type information a language can be made more expressive! Overloading, dynamic binding!

3. Run-time efficiency

- Information from types can be used by the compiler to generate better code!
- Run-time checks can be avoided with static (compile-time) checking.

The purpose of type systems

1. Run-time safety

The compiler should eliminate as many run-time errors as it can using type-checking techniques.

2. Increased expressiveness

By using type information a language can be made more expressive! Overloading, dynamic binding!

3. Run-time efficiency

- Information from types can be used by the compiler to generate better code!
- Run-time checks can be avoided with static (compile-time) checking.

The purpose of type systems

1. Run-time safety

The compiler should eliminate as many run-time errors as it can using type-checking techniques.

2. Increased expressiveness

By using type information a language can be made more expressive! Overloading, dynamic binding!

3. Run-time efficiency

- Information from types can be used by the compiler to generate better code!
- Run-time checks can be avoided with static (compile-time) checking.

The purpose of type systems

1. Run-time safety

The compiler should eliminate as many run-time errors as it can using type-checking techniques.

2. Increased expressiveness

By using type information a language can be made more expressive! Overloading, dynamic binding!

3. Run-time efficiency

- Information from types can be used by the compiler to generate better code!
- Run-time checks can be avoided with static (compile-time) checking.

An example in Java

Example

```
class MovingPoint{
    protected int x, y;
    public MovingPoint(int a, int b){
        x=a;y=b;
    }
    public void move(){}
    public void report(){
        System.out.println("MP("+x+", "+y+")");
    }
}
```

An example in Java

Example

```
class Uniform extends MovingPoint{
    protected int vx, vy;
    public Uniform(int a, int b, int cx, int cy){
      super(a,b);vx=cx;vy=cy;
    }
    public void move(){
      x=x+vx;
      y=y+vy;
    }
    public void report(){
      System.out.println("U("+x+", "+y+")");
    }
}
```

An example in Java

Example

```
class Accelerated extends Uniform{
    protected int ax, ay;
    public Accelerated(int a, int b, int cx, int cy, int dx
      super(a,b,cx,cy);ax=dx;ay=dy;
    }
    public void move(){
      x=x+vx;
      y=y+vy;
      vx=vx+ax;
      vy=vy+ay;
    }
    public void report(){
      System.out.println("A("+x+", "+y+")");
    }
}
```

Run-time safety?

```
public static void main(String[] cmdLine){
  List  mps = new ArrayList();
  mps.add(new MovingPoint(0,0));
  mps.add(new Uniform(0,0,10,10));
  mps.add("This is not right!");
  for(int i = 0; i<10;i++){
    Iterator iter = mps.iterator();
    while(iter.hasNext()){
      MovingPoint mp = (MovingPoint)iter.next();
      mp.move();
      mp.report();
    }
    System.out.println();
  }
}
```

Run-time safety?

```
lecture6 > java OldPoints
MP(0, 0)
U(10, 10)
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
        at OldPoints.main(OldPoints.java:10)
lecture6 >
```

Run-time safety *"Well-typed programs never go wrong."*

```
class Points{
  public static void main(String[] cmdLine){
    List<MovingPoint> mps = new ArrayList<MovingPoint>();
    mps.add(new MovingPoint(0,0));
    mps.add(new Uniform(0,0,10,10));
    mps.add(new Accelerated(0,0,10,10,3,6));
    for(int i = 0; i<10;i++){
      for(MovingPoint mp : mps){
        mp.move();
        mp.report();
      }
      System.out.println();
    }
  }
}
```

Run-time safety *"Well-typed programs never go wrong."*

```
lecture6 > java Points
MP(0, 0)
U(10, 10)
A(10, 10)

MP(0, 0)
U(20, 20)
A(23, 26)

MP(0, 0)
U(30, 30)
A(39, 48)

MP(0, 0)
U(40, 40)
A(58, 76)
```

Expressiveness

```
class Points{
  public static void main(String[] cmdLine){
    List<MovingPoint> mps = new ArrayList<MovingPoint>();
    mps.add(new MovingPoint(0,0));
    mps.add(new Uniform(0,0,10,10));
    mps.add(new Accelerated(0,0,10,10,3,6));
    for(int i = 0; i<10;i++){
      for(MovingPoint mp : mps){
        mp.move();
        mp.report();
      }
      System.out.println();
    }
  }
}
```

Run-time efficiency

a	b	a+b	assembler
`integer`	`integer`	`integer`	`iADD` r_a r_b $\Rightarrow$ r_{a+b}
`double`	`double`	`double`	`dADD` r_a r_b $\Rightarrow$ r_{a+b}

The compiler can generate right code! In older languages the compiler generated code that made tests on the types of the arguments during run-time!

Type Checking

Language processors include a

Type checker

- That infers the type of expressions in the source program
- and controls whether the source program complies with the type system of the language.
- Programs that do not comply are not processed further, they are considered meaningless.

Type checking during compilation (*static type checking*) has as consequences

- efficiency of target execution!
- guarantees that certain runtime errors cannot occur!

Type Checking

Language processors include a

Type checker

- That infers the type of expressions in the source program
- and controls whether the source program complies with the type system of the language.
- Programs that do not comply are not processed further, they are considered meaningless.

Type checking during compilation (*static type checking*) has as consequences

- efficiency of target execution!
- guarantees that certain runtime errors cannot occur!

Type Checking

Language processors include a

Type checker

- That infers the type of expressions in the source program
- and controls whether the source program complies with the type system of the language.
- Programs that do not comply are not processed further, they are considered meaningless.

Type checking during compilation (*static type checking*) has as consequences

- efficiency of target execution!
- guarantees that certain runtime errors cannot occur!

Type Checking

Language processors include a

Type checker

- That infers the type of expressions in the source program
- and controls whether the source program complies with the type system of the language.
- Programs that do not comply are not processed further, they are considered meaningless.

Type checking during compilation (*static type checking*) has as consequences

- efficiency of target execution!
- guarantees that certain runtime errors cannot occur!

Type Checking

Language processors include a

Type checker

- That infers the type of expressions in the source program
- and controls whether the source program complies with the type system of the language.
- Programs that do not comply are not processed further, they are considered meaningless.

Type checking during compilation (*static type checking*) has as consequences

- efficiency of target execution!
- guarantees that certain runtime errors cannot occur!

Type Checking

Language processors include a

Type checker

- That infers the type of expressions in the source program
- and controls whether the source program complies with the type system of the language.
- Programs that do not comply are not processed further, they are considered meaningless.

Type checking during compilation (*static type checking*) has as consequences

- efficiency of target execution!
- guarantees that certain runtime errors cannot occur!

Type Checking

Language processors include a

Type checker

- That infers the type of expressions in the source program
- and controls whether the source program complies with the type system of the language.
- Programs that do not comply are not processed further, they are considered meaningless.

Type checking during compilation (*static type checking*) has as consequences

- efficiency of target execution!
- guarantees that certain runtime errors cannot occur!

Type Systems

Almost all computer languages include a

Type System

that prescribes

- Some *primitive types*.
- How *new types* can be introduced.
- What the *values of each type* are.
- What the *legal operations* for each type are (or what the type of each operation is).
- Type constraints for the constructs of the language.

Type Systems

Almost all computer languages include a

Type System

that prescribes

- Some *primitive types*.
- How *new types* can be introduced.
- What the *values of each type* are.
- What the *legal operations* for each type are (or what the type of each operation is).
- Type constraints for the constructs of the language.

Type Systems

Almost all computer languages include a

Type System

that prescribes

- Some *primitive types*.
- How *new types* can be introduced.
- What the *values of each type* are.
- What the *legal operations* for each type are (or what the type of each operation is).
- Type constraints for the constructs of the language.

Type Systems

Almost all computer languages include a

Type System

that prescribes

- Some *primitive types*.
- How *new types* can be introduced.
- What the *values of each type* are.
- What the *legal operations* for each type are (or what the type of each operation is).
- Type constraints for the constructs of the language.

Type Systems

Almost all computer languages include a

Type System

that prescribes

- Some *primitive types*.
- How *new types* can be introduced.
- What the *values of each type* are.
- What the *legal operations* for each type are (or what the type of each operation is).
- Type constraints for the constructs of the language.

Type Systems

Almost all computer languages include a

Type System

that prescribes

- Some *primitive types*.
- How *new types* can be introduced.
- What the *values of each type* are.
- What the *legal operations* for each type are (or what the type of each operation is).
- Type constraints for the constructs of the language.

Type system for `minijava` - Primitive types

```
boolean
```

```
int
```

```
int[]
```

Type system for `minijava` - Primitive types

```
boolean
```

```
int
```

```
int[]
```

Type system for `minijava` - Primitive types

```
boolean
```

```
int
```

```
int[]
```

Type system for `minijava` - New types

Every class declaration introduces a new type.

Type system for `minijava` - Values

`boolean`

`true`, `false`

`int`

`...`, `-3`, `-2`, `-1`, `0`, `1`, `2`, `3`, `...`

`int[]`

`new int[`*`expWithIntegerValue`*`]`

`AClassName`

`new AClassName()`, `this`

Type system for `minijava` - Values

`boolean`

`true`, `false`

`int`

`...`, `-3`, `-2`, `-1`, `0`, `1`, `2`, `3`, `...`

`int[]`

`new int[`*`expWithIntegerValue`*`]`

`AClassName`

`new AClassName()`, `this`

Type system for `minijava` - Values

`boolean`

`true`, `false`

`int`

`...`, `-3`, `-2`, `-1`, `0`, `1`, `2`, `3`, `...`

`int[]`

`new int[`*`expWithIntegerValue`*`]`

`AClassName`

`new AClassName()`, `this`

Type system for `minijava` - Values

`boolean`

`true`, `false`

`int`

`...`, `-3`, `-2`, `-1`, `0`, `1`, `2`, `3`, `...`

`int[]`

`new int[`*`expWithIntegerValue`*`]`

`AClassName`

`new AClassName()`, `this`

Type system for `minijava` - Operations

`int`	`<`	`int`	`boolean`
`boolean`	`&`	`boolean`	`boolean`
	`!`	`boolean`	`boolean`
`int`	`+`	`int`	`int`
`int`	`-`	`int`	`int`
`int`	`*`	`int`	`int`
`int[]`	`.length`		`int`
`int[]`	`[int]`		`int`
`obj`	`.mthd(args)`		`T` *(return type of mthd)*

Type system for `minijava` - The language constructs

The variable or formal declaration

```
type id
```

makes

`id` an expression of type `type`.

```
if (e) s1 else s2
```

requires

`e` to be of type `boolean` and `s1` and `s2` to be well typed.

```
while (e) s
```

requires

`e` to be of type `boolean` and `s` to be well typed.

```
System.out.println(e)
```

requires

`e` to be of type `int`.

Type system for `minijava` - The language constructs

The variable or formal declaration

```
type id
```

makes

`id` an expression of type `type`.

```
if (e) s1 else s2
```

requires

`e` to be of type `boolean` and `s1` and `s2` to be well typed.

```
while (e) s
```

requires

`e` to be of type `boolean` and `s` to be well typed.

```
System.out.println(e)
```

requires

`e` to be of type `int`.

Type system for `minijava` - The language constructs

The variable or formal declaration

`type id`

makes

`id` an expression of type `type`.

`if (e) s1 else s2`

requires

`e` to be of type `boolean` and `s1` and `s2` to be well typed.

`while (e) s`

requires

`e` to be of type `boolean` and `s` to be well typed.

`System.out.println(e)`

requires

`e` to be of type `int`.

Type system for `minijava` - The language constructs

The variable or formal declaration

```
type id
```

makes

`id` an expression of type `type`.

```
if (e) s1 else s2
```

requires

`e` to be of type `boolean` and `s1` and `s2` to be well typed.

```
while (e) s
```

requires

`e` to be of type `boolean` and `s` to be well typed.

```
System.out.println(e)
```

requires

`e` to be of type `int`.

Type system for `minijava` - The language constructs

The variable or formal declaration

```
type id
```

makes

`id` an expression of type `type`.

```
if (e) s1 else s2
```

requires

`e` to be of type `boolean` and `s1` and `s2` to be well typed.

```
while (e) s
```

requires

`e` to be of type `boolean` and `s` to be well typed.

```
System.out.println(e)
```

requires

`e` to be of type `int`.

Type system for `minijava` - The language constructs

The variable or formal declaration

```
type id
```

makes

`id` an expression of type `type`.

```
while (e) s
```

requires

`e` to be of type `boolean` and `s` to be well typed.

```
if (e) s1 else s2
```

requires

`e` to be of type `boolean` and `s1` and `s2` to be well typed.

```
System.out.println(e)
```

requires

`e` to be of type `int`.

Type system for `minijava` - The language constructs

The variable or formal declaration

```
type id
```

makes

`id` an expression of type `type`.

```
while (e) s
```

requires

`e` to be of type `boolean` and `s` to be well typed.

```
if (e) s1 else s2
```

requires

`e` to be of type `boolean` and `s1` and `s2` to be well typed.

```
System.out.println(e)
```

requires

`e` to be of type `int`.

Type system for `minijava` - The language constructs

In

```
id = e
```

the types of `id` and `e` must *"coincide"*.

In

```
id[e1] = e2
```

`id` must be of type `int[]` and `e1` and `e2` must be of type `int`.

In

```
public t m(formalArguments){
 decs
 stmts
 return e;
}
```

the type of `e` and `t` must *"coincide"*.

Type system for `minijava` - The language constructs

In

```
id = e
```

the types of `id` and `e` must *"coincide"*.

In

```
id[e1] = e2
```

`id` must be of type `int[]` and `e1` and `e2` must be of type `int`.

In

```
public t m(formalArguments){
 decs
 stmts
 return e;
}
```

the type of `e` and `t` must *"coincide"*.

Type system for `minijava` - The language constructs

In

```
id = e
```

the types of `id` and `e` must *"coincide"*.

In

```
id[e1] = e2
```

`id` must be of type `int[]` and `e1` and `e2` must be of type `int`.

In

```
public t m(formalArguments){
 decs
 stmts
 return e;
}
```

the type of `e` and `t` must *"coincide"*.

Type system for `minijava` - The language constructs

In

```
id = e
```

the types of `id` and `e` must *"coincide"*.

In

```
id[e1] = e2
```

`id` must be of type `int[]` and `e1` and `e2` must be of type `int`.

Type system for `minijava` - The language constructs

In

```
id = e
```

the types of `id` and `e` must *"coincide"*.

In

```
id[e1] = e2
```

`id` must be of type `int[]` and `e1` and `e2` must be of type `int`.

In

```
public t m(formalArguments){
 decs
 stmts
 return e;
}
```

the type of `e` and `t` must *"coincide"*.

Type system for `minijava` - The language constructs

In

```
id = e
```

the types of `id` and `e` must *"coincide"*.

In

```
id[e1] = e2
```

`id` must be of type `int[]` and `e1` and `e2` must be of type `int`.

In

```
public t m(formalArguments){
 decs
 stmts
 return e;
}
```

the type of `e` and `t` must *"coincide"*.

Type system for `minijava` - The language constructs

In a method call

```
e.m(actualArguments)
```

the types of `actualArguments` must *"coincide"* with the types of the `formalArguments` in the method declaration and `e` must be have as type a class where the method `m` is declared.

Subtypes

In *minijava* the class declaration

`class B extends A`

introduces a type `B` that is said to be a *subtype* of `A`. Meaning that whenever an `A` is needed a `B` can be used.

In other words, all expressions of type `B` are also expressions of type `A`.

Subtypes

In *minijava* the class declaration

`class B extends A`

introduces a type `B` that is said to be a *subtype* of `A`. Meaning that whenever an `A` is needed a `B` can be used.

In other words, all expressions of type `B` are also expressions of type `A`.

Subtypes

In *minijava* the class declaration

`class B extends A`

introduces a type `B` that is said to be a *subtype* of `A`. Meaning that whenever an `A` is needed a `B` can be used.

In other words, all expressions of type `B` are also expressions of type `A`.

Subtypes

In *minijava* the class declaration

`class B extends A`

introduces a type `B` that is said to be a *subtype* of `A`. Meaning that whenever an `A` is needed a `B` can be used.

In other words, all expressions of type `B` are also expressions of type `A`.

Subtypes

In *minijava* the class declaration

`class B extends A`

introduces a type `B` that is said to be a *subtype* of `A`. Meaning that whenever an `A` is needed a `B` can be used.

In other words, all expressions of type `B` are also expressions of type `A`.

Subtypes

In *minijava* the class declaration
`class B extends A`
introduces a type `B` that is said to be a *subtype* of `A`. Meaning that whenever an `A` is needed a `B` can be used.
In other words, all expressions of type `B` are also expressions of type `A`.

Subtypes

In *minijava* the class declaration
`class B extends A`
introduces a type `B` that is said to be a *subtype* of `A`. Meaning that whenever an `A` is needed a `B` can be used.
In other words, all expressions of type `B` are also expressions of type `A`.

A short note on the type system of Java

- The type system of *minijava* is just a reduced version of that of JAVA. We will now look at some examples in JAVA to explore some of the consequences (for the programmer) of this type system.
- The possibility of defining subtypes together with the class system favor *factorization* of common code. The methods that are common to some classes are programmed once in a class that is then extended.
- This can be exploited also when defining *generic* classes, like the common datastructures for *collections* in order to define the algorithms once and for all, despite the type of the elements.

A short note on the type system of Java

- The type system of *minijava* is just a reduced version of that of JAVA. We will now look at some examples in JAVA to explore some of the consequences (for the programmer) of this type system.
- The possibility of defining subtypes together with the class system favor *factorization* of common code. The methods that are common to some classes are programmed once in a class that is then extended.
- This can be exploited also when defining *generic* classes, like the common datastructures for *collections* in order to define the algorithms once and for all, despite the type of the elements.

A short note on the type system of Java

- The type system of *minijava* is just a reduced version of that of JAVA. We will now look at some examples in JAVA to explore some of the consequences (for the programmer) of this type system.
- The possibility of defining subtypes together with the class system favor *factorization* of common code. The methods that are common to some classes are programmed once in a class that is then extended.
- This can be exploited also when defining *generic* classes, like the common datastructures for *collections* in order to define the algorithms once and for all, despite the type of the elements.

A short note on the type system of Java

- In JAVA all classes extend the class `Object`
- All classes in the the collections framework are defined in terms of this.

Example

```
class Stack
 public Stack()
 public boolean empty()
 public Object peek()
 public Object pop()
 public Object push(Object item)
```

- We can use it in our programs to stack any kind of objects!
- Even objects of classes we define ourselves and were unknown to the developer of `Stack`!

A short note on the type system of Java

- In JAVA all classes extend the class `Object`
- All classes in the the collections framework are defined in terms of this.

Example

```
class Stack
 public Stack()
 public boolean empty()
 public Object peek()
 public Object pop()
 public Object push(Object item)
```

- We can use it in our programs to stack any kind of objects!
- Even objects of classes we define ourselves and were unknown to the developer of `Stack`!

A short note on the type system of Java

- In JAVA all classes extend the class `Object`
- All classes in the the collections framework are defined in terms of this.

Example

```
class Stack
 public Stack()
 public boolean empty()
 public Object peek()
 public Object pop()
 public Object push(Object item)
```

- We can use it in our programs to stack any kind of objects!
- Even objects of classes we define ourselves and were unknown to the developer of `Stack`!

A short note on the type system of Java

- In JAVA all classes extend the class `Object`
- All classes in the the collections framework are defined in terms of this.

Example

```
class Stack
 public Stack()
 public boolean empty()
 public Object peek()
 public Object pop()
 public Object push(Object item)
```

- We can use it in our programs to stack any kind of objects!
- Even objects of classes we define ourselves and were unknown to the developer of `Stack`!

A short note on the type system of Java

- In JAVA all classes extend the class `Object`
- All classes in the the collections framework are defined in terms of this.

Example

```
class Stack
 public Stack()
 public boolean empty()
 public Object peek()
 public Object pop()
 public Object push(Object item)
```

- We can use it in our programs to stack any kind of objects!
- Even objects of classes we define ourselves and were unknown to the developer of `Stack`!

A short note on the type system of Java

- However this has some *complications* . . . .
- When we use stacks in our programs we are supposed to fill them with objects of the same type so that we can do reasonable things with them!

Example

```
void moveAll(Stack theShapes){
    while (!theShapes.empty())
        ((Shape)theShapes.pop()).move();
}
```

- We cannot express with the type system that the argument to `moveAll` is a stack of `Shapes`.
- We are forced to use *cohercion* to be able, for instance, to `move()` the shapes in the stack!

A short note on the type system of Java

- However this has some *complications* . . . .
- When we use stacks in our programs we are supposed to fill them with objects of the same type so that we can do reasonable things with them!

Example

```
void moveAll(Stack theShapes){
    while (!theShapes.empty())
        ((Shape)theShapes.pop()).move();
}
```

- We cannot express with the type system that the argument to `moveAll` is a stack of `Shapes`.
- We are forced to use *cohercion* to be able, for instance, to `move()` the shapes in the stack!

A short note on the type system of Java

- However this has some *complications* . . . .
- When we use stacks in our programs we are supposed to fill them with objects of the same type so that we can do reasonable things with them!

Example

```
void moveAll(Stack theShapes){
    while (!theShapes.empty())
       ((Shape)theShapes.pop()).move();
}
```

- We cannot express with the type system that the argument to `moveAll` is a stack of `Shapes`.
- We are forced to use *cohercion* to be able, for instance, to `move()` the shapes in the stack!

A short note on the type system of Java

- However this has some *complications* . . . .
- When we use stacks in our programs we are supposed to fill them with objects of the same type so that we can do reasonable things with them!

Example

```
void moveAll(Stack theShapes){
    while (!theShapes.empty())
       ((Shape)theShapes.pop()).move();
}
```

- We cannot express with the type system that the argument to `moveAll` is a stack of `Shapes`.
- We are forced to use *cohercion* to be able, for instance, to `move()` the shapes in the stack!

A short note on the type system of Java

- However this has some *complications* . . . .
- When we use stacks in our programs we are supposed to fill them with objects of the same type so that we can do reasonable things with them!

Example

```
void moveAll(Stack theShapes){
    while (!theShapes.empty())
       ((Shape)theShapes.pop()).move();
}
```

- We cannot express with the type system that the argument to `moveAll` is a stack of `Shapes`.
- We are forced to use *cohercion* to be able, for instance, to `move()` the shapes in the stack!

A short note on the type system of Java

- Moreover, there is no way of guarding against type violations during runtime! (which result in runtime errors!)
- We can call the method `moveAll` with a Stack containing objects of any class and even with objects of different classes!

Example

```
Stack s = new Stack();
s.push(new Circle(3,3,10);)
s.push(‘‘hej’’);
s.push(new Integer(2));
moveAll(s);
```

- So, the way we used subtyping for defining `Stack` is somewhat an incorrect use of subtyping. But this is what we have in Java!

A short note on the type system of Java

- Moreover, there is no way of guarding against type violations during runtime! (which result in runtime errors!)
- We can call the method `moveAll` with a Stack containing objects of any class and even with objects of different classes!

Example

```
Stack s = new Stack();
s.push(new Circle(3,3,10);)
s.push(``hej'');
s.push(new Integer(2));
moveAll(s);
```

- So, the way we used subtyping for defining `Stack` is somewhat an incorrect use of subtyping. But this is what we have in Java!

A short note on the type system of Java

- Moreover, there is no way of guarding against type violations during runtime! (which result in runtime errors!)
- We can call the method `moveAll` with a Stack containing objects of any class and even with objects of different classes!

Example

```
Stack s = new Stack();
s.push(new Circle(3,3,10);)
s.push(``hej'');
s.push(new Integer(2));
moveAll(s);
```

- So, the way we used subtyping for defining `Stack` is somewhat an incorrect use of subtyping. But this is what we have in Java!

A short note on the type system of Java

- Moreover, there is no way of guarding against type violations during runtime! (which result in runtime errors!)
- We can call the method `moveAll` with a Stack containing objects of any class and even with objects of different classes!

Example

```
Stack s = new Stack();
s.push(new Circle(3,3,10);)
s.push(``hej'');
s.push(new Integer(2));
moveAll(s);
```

- So, the way we used subtyping for defining `Stack` is somewhat an incorrect use of subtyping. But this is what we have in Java!

Beyond

- This is a well known problem that has been solved and implemented a long time ago!
- In the context of other programming languages since the 70s
- In the context of JAVA since the late 90s. (GJ and PolyJ)
- There is now a new release of JAVA strongly based on GJ.

Parameterized classes

- A class can be made to depend on parameters that can be instantiated with types
- The fields and methods of these classes can be typed using the parameters

- This does not imply modifications to the JVM because it is implemented by translating to JAVA

Beyond

- This is a well known problem that has been solved and implemented a long time ago!
- In the context of other programming languages since the 70s
- In the context of JAVA since the late 90s. (GJ and PolyJ)
- There is now a new release of JAVA strongly based on GJ.

Parameterized classes

- A class can be made to depend on parameters that can be instantiated with types
- The fields and methods of these classes can be typed using these parameters

- This does not imply modifications to the JVM because it is implemented by translating to JAVA

Beyond

- This is a well known problem that has been solved and implemented a long time ago!
- In the context of other programming languages since the 70s
- In the context of JAVA since the late 90s. (GJ and PolyJ)
- There is now a new release of JAVA strongly based on GJ.

Parameterized classes

- A class can be parameterized with one or more types that can be instantiated arbitrarily.
- The fields and methods of parameterized classes can refer to the parameter types.

- This does not imply modifications to the JVM because it is implemented by translating to JAVA

Beyond

- This is a well known problem that has been solved and implemented a long time ago!
- In the context of other programming languages since the 70s
- In the context of JAVA since the late 90s. (GJ and PolyJ)
- There is now a new release of JAVA strongly based on GJ.

Parameterized classes

- A class can be made to depend on parameters that can be instantiated with types
- The fields and methods of these classes can be typed using these parameters

- This does not imply modifications to the JVM because it is implemented by translating to JAVA

Beyond

- This is a well known problem that has been solved and implemented a long time ago!
- In the context of other programming languages since the 70s
- In the context of JAVA since the late 90s. (GJ and PolyJ)
- There is now a new release of JAVA strongly based on GJ.

Parameterized classes

- A class can be made to depend on parameters that can be instantiated with types
- The fields and methods of these classes can be typed using these parameters

- This does not imply modifications to the JVM because it is implemented by translating to JAVA

Beyond

- This is a well known problem that has been solved and implemented a long time ago!
- In the context of other programming languages since the 70s
- In the context of JAVA since the late 90s. (GJ and PolyJ)
- There is now a new release of JAVA strongly based on GJ.

Parameterized classes

- A class can be made to depend on parameters that can be instantiated with types
- The fields and methods of these classes can be typed using these parameters

- This does not imply modifications to the JVM because it is implemented by translating to JAVA

Beyond

- This is a well known problem that has been solved and implemented a long time ago!
- In the context of other programming languages since the 70s
- In the context of JAVA since the late 90s. (GJ and PolyJ)
- There is now a new release of JAVA strongly based on GJ.

Parameterized classes

- A class can be made to depend on parameters that can be instantiated with types
- The fields and methods of these classes can be typed using these parameters

- This does not imply modifications to the JVM because it is implemented by translating to JAVA

Beyond

- This is a well known problem that has been solved and implemented a long time ago!
- In the context of other programming languages since the 70s
- In the context of JAVA since the late 90s. (GJ and PolyJ)
- There is now a new release of JAVA strongly based on GJ.

Parameterized classes

- A class can be made to depend on parameters that can be instantiated with types
- The fields and methods of these classes can be typed using these parameters

- This does not imply modifications to the JVM because it is implemented by translating to JAVA

Beyond

In java 1.5 examples above will instead look as

Example

```
class Stack<E>
    public Stack()
    public boolean empty()
    public E peek()
    public E pop()
    public E push(E item)
```

Example

```
void moveAll(Stack<Shape> theShapes){
    while (!theShapes.empty())
        (theShapes.pop()).move();
}
```

Beyond

In java 1.5 examples above will instead look as

Example

```
class Stack<E>
    public Stack()
    public boolean empty()
    public E peek()
    public E pop()
    public E push(E item)
```

Example

```
void moveAll(Stack<Shape> theShapes){
    while (!theShapes.empty())
       (theShapes.pop()).move();
}
```