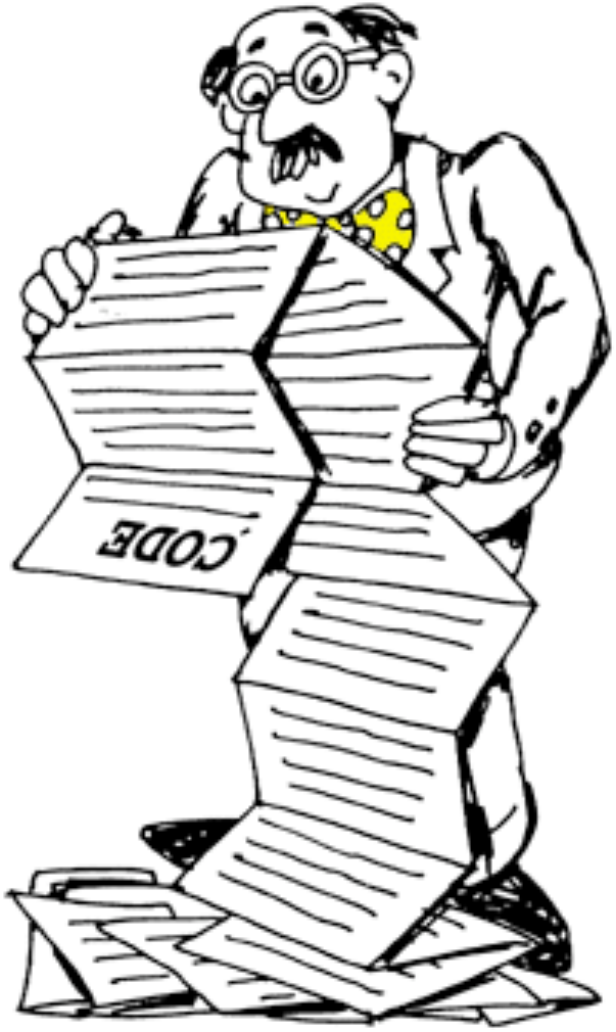
Computer Languages

`minijava` - Concrete & Abstract syntax

1. Parsing
2. Syntactic categories
3. Concrete syntax
4. Abstract syntax
5. Composite pattern

February 4

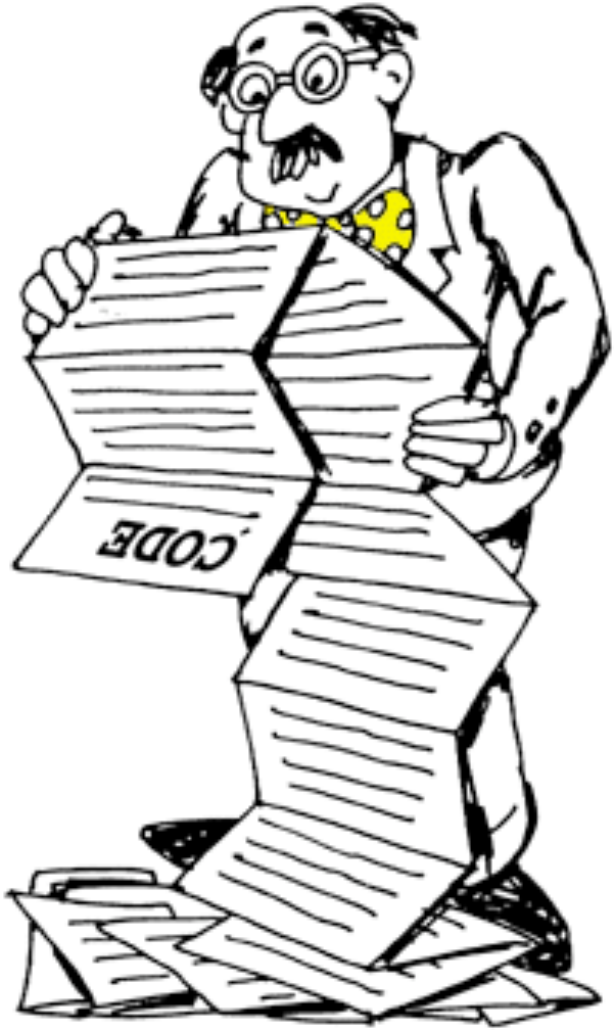
Overview of a compiler



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.

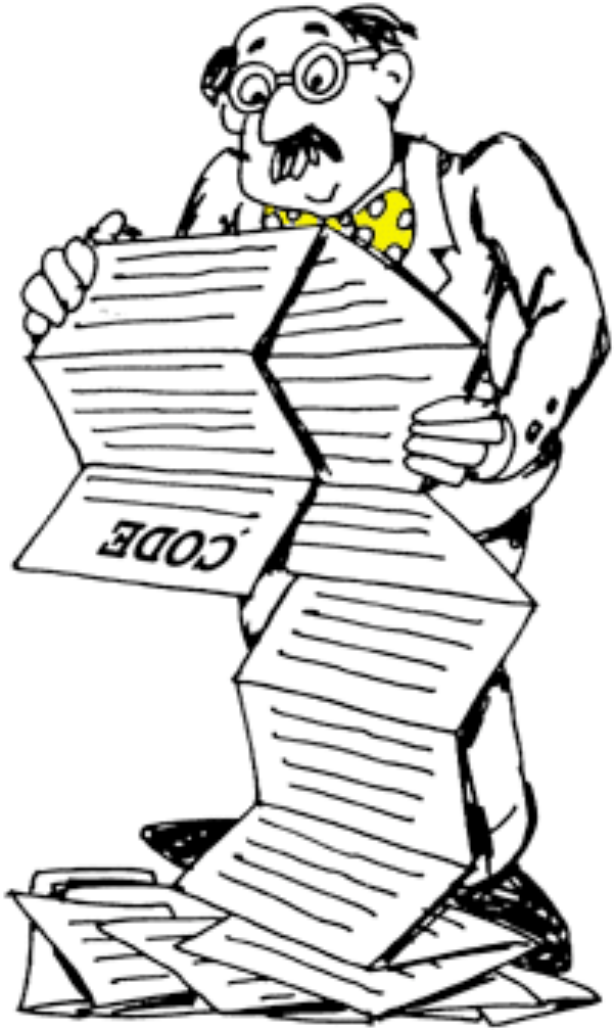
Overview of a compiler



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.

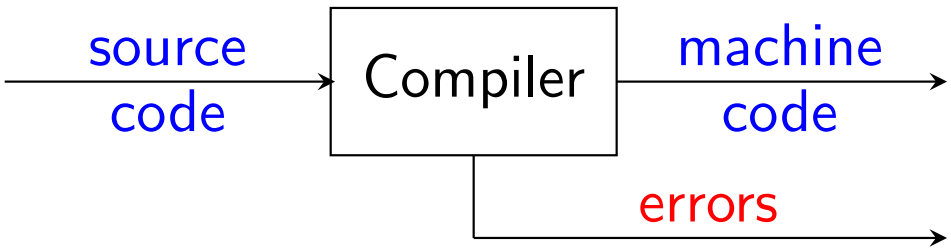
Overview of a compiler



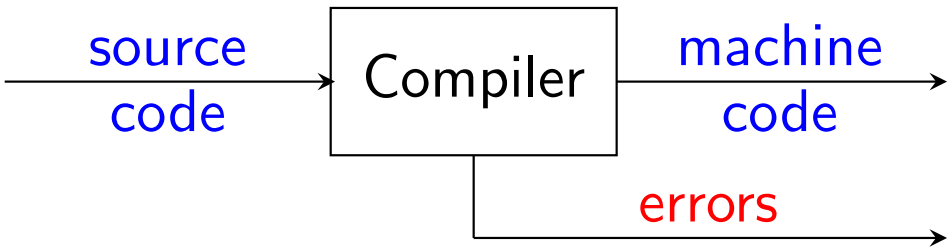
The compiler has to

- 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

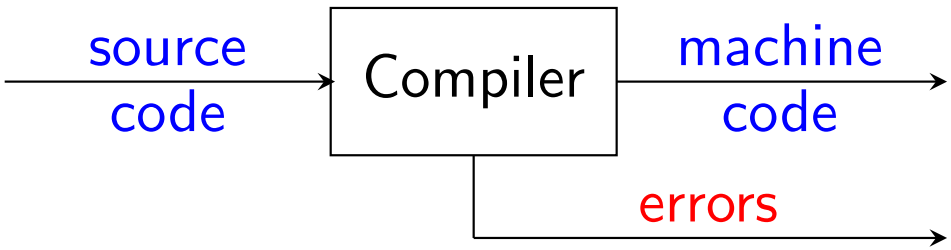


Overview of a compiler



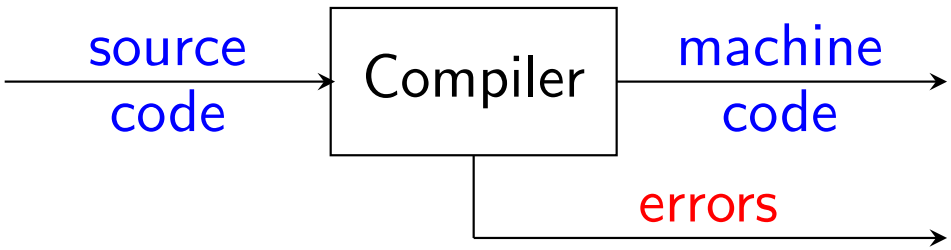
- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!

Overview of a compiler



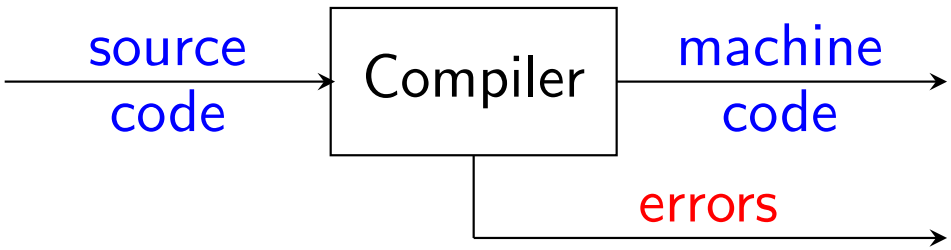
- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!

Overview of a compiler



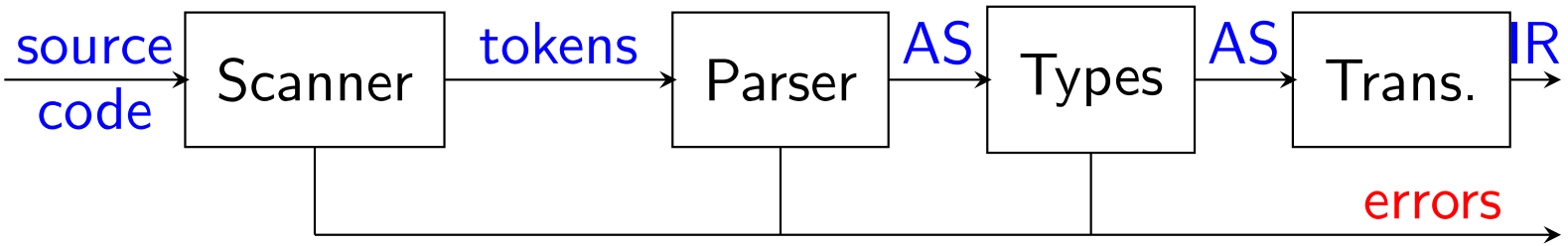
- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!

Overview of a compiler



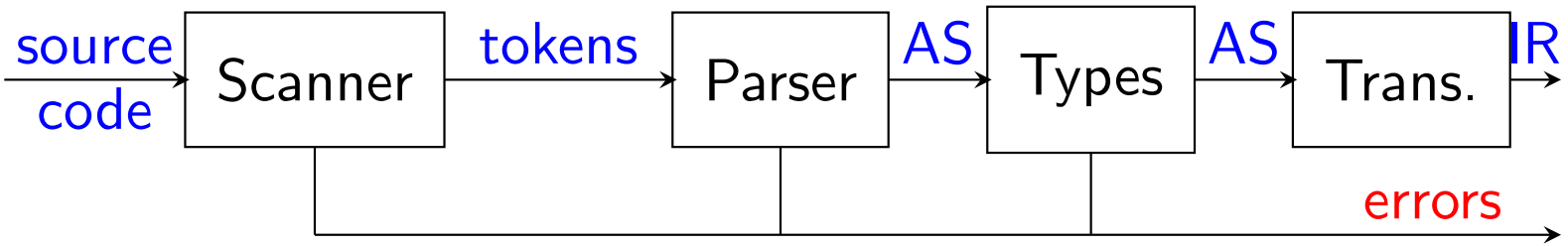
- Has to distinguish correct from incorrect programs (has to understand!)
- Has to generate correct machine code!
- Has to organize memory for variables and instructions!
- Has to agree with OS on the form of object code!

The Front End (analysis phase)



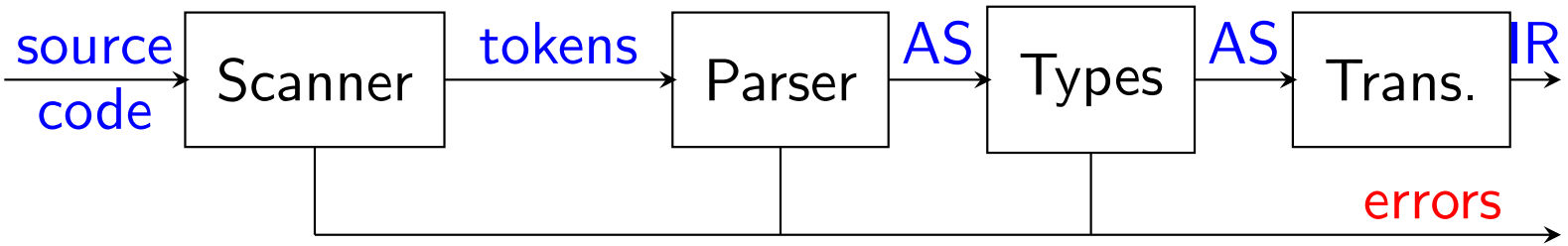
- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed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

The Front End (analysis phase)



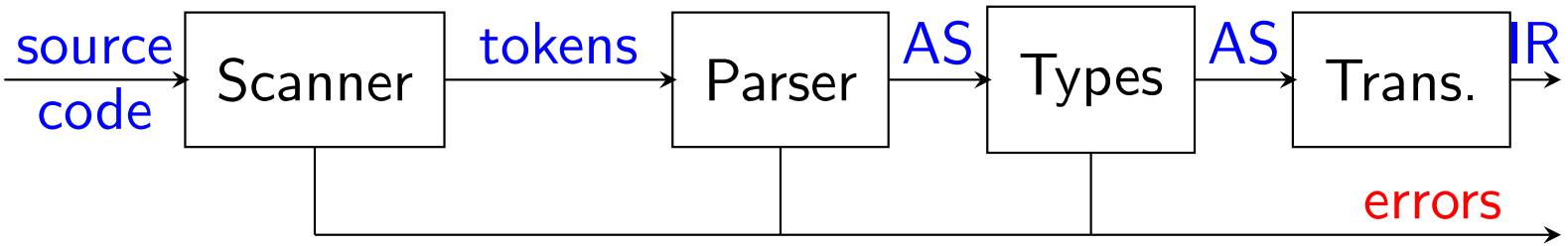
- The Scanner (lexical analyzer) transforms a sequence of characters (source code) into a sequence of tokens: a representation of the *lexemes* of the language.
- The Parser (syntactical analyzer) takes the sequence of tokens and generates a tree representation, the Abstract Syntax.
- This tree is analyzed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

Plan for today

Syntactic categories

What kind of phrases are used in a programming language? What components do they have?

Concrete syntax

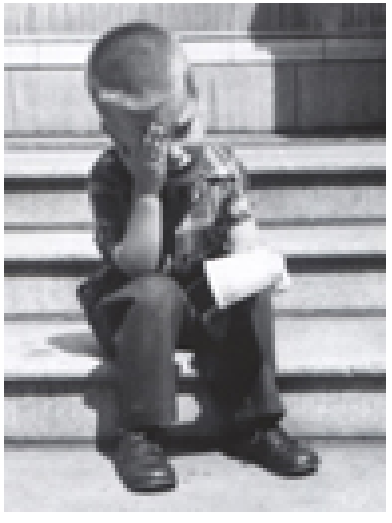
How do the phrases look like? What words, separators, punctuations are used?

Abstract syntax

What data structure can be used to store the program so that the structure is evident?

Composite pattern

How can we implement this data structure in Java?



Plan for today

Syntactic categories

What kind of phrases are used in a programming language? What components do they have?

Concrete syntax

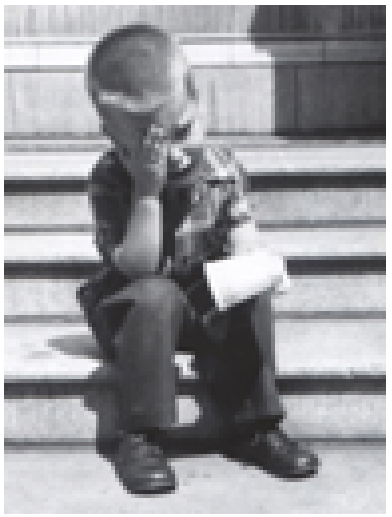
How do the phrases look like? What words, separators, punctuations are used?

Abstract syntax

What data structure can be used to store the program so that the structure is evident?

Composite pattern

How can we implement this data structure in Java?



Plan for today

Syntactic categories

What kind of phrases are used in a programming language? What components do they have?

Concrete syntax

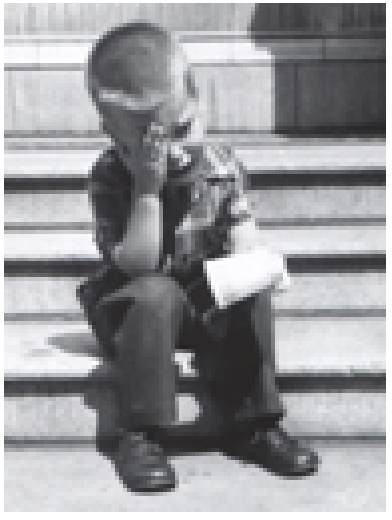
How do the phrases look like? What words, separators, punctuations are used?

Abstract syntax

What data structure can be used to store the program so that the structure is evident?

Composite pattern

How can we implement this data structure in Java?



Plan for today

Syntactic categories

What kind of phrases are used in a programming language? What components do they have?

Concrete syntax

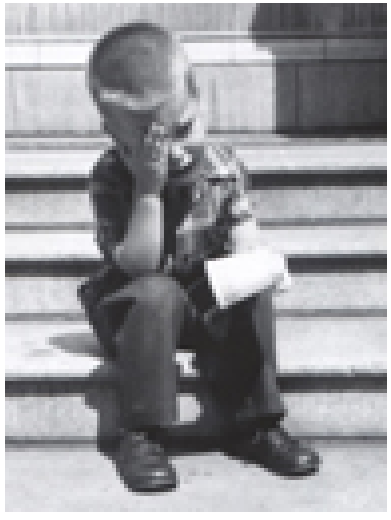
How do the phrases look like? What words, separators, punctuations are used?

Abstract syntax

What data structure can be used to store the program so that the structure is evident?

Composite pattern

How can we implement this data structure in Java?



Plan for today

Syntactic categories

What kind of phrases are used in a programming language? What components do they have?

Concrete syntax

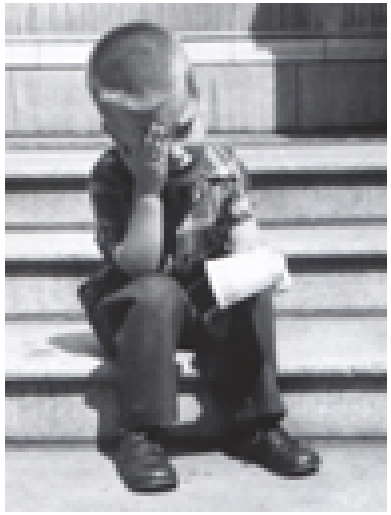
How do the phrases look like? What words, separators, punctuations are used?

Abstract syntax

What data structure can be used to store the program so that the structure is evident?

Composite pattern

How can we implement this data structure in Java?



Syntactic categories in programming languages

Example

```
class Small{
    public static void main(String args[]){
        System.out.println("Hello "+args[0]);
    }
}
```

Example

```
resource small()
  string[10] user
  getarg(1,user)
  write("Hello",user)
end
```

Example

```
import System


main =
   do (user:_) <- getArgs
      print("Hello " ++ user)
```

Syntactic categories in programming languages

Example

```
class Small{
    public static void main(String args[]){
        System.out.println("Hello "+args[0]);
    }
}
```

Example

```
resource small()
  string[10] user
  getarg(1,user)
  write("Hello",user)
end
```

Example

```
import System


main =
   do (user:_) <- getArgs
      print("Hello " ++ user)
```

Syntactic categories in programming languages

Example

```
class Small{
    public static void main(String args[]){
        System.out.println("Hello "+args[0]);
    }
}
```

Example

```
resource small()
  string[10] user
  getarg(1,user)
  write("Hello",user)
end
```

Example

```
import System


main =
   do (user:_) <- getArgs
      print("Hello " ++ user)
```

Syntactic categories in programming languages

Example

```
class Small{
    public static void main(String args[]){
        System.out.println("Hello "+args[0]);
    }
}
```

Example

```
resource small()
  string[10] user
  getarg(1,user)
  write("Hello",user)
end
```

Example

```
import System

main =
   do (user:_) <- getArgs
      print("Hello " ++ user)
```

Syntactic categories in programming languages

What is a Program?

- In **Java** and **minijava**: a class with a static main method!
  - and some class declarations!
- In C and C++: a main function (a static method!)!
  - and some function or class declarations!
- In other languages it could be something else.

Syntactic categories in programming languages

What is a Program?

- In **Java** and **minijava**: a class with a static main method!
  - and some class declarations!
- In C and C++: a main function (a static method!)!
  - and some function or class declarations!
- In other languages it could be something else.

Syntactic categories in programming languages

What is a Program?

- In **Java** and **minijava**: a class with a static main method!
  - and some class declarations!
- In C and C++: a main function (a static method!)!
  - and some function or class declarations!
- In other languages it could be something else.

Syntactic categories in programming languages

What is a Program?

- In **Java** and **minijava**: a class with a static main method!
  - and some class declarations!
- In C and C++: a main function (a static method!)!
  - and some function or class declarations!
- In other languages it could be something else.

Syntactic categories in programming languages

Declarations

What we use to build abstractions: we give names to entities.

Expressions

Entities that have a value. Expressions usually have an associated type.

Statements

Entities that change state.

Some programming languages have no statements (*Haskell*). In some languages statements and expressions are mixed up (*C*). Different programming paradigms support different kinds of values and abstractions.

Syntactic categories in programming languages

Declarations

What we use to build abstractions: we give names to entities.

Expressions

Entities that have a value. Expressions usually have an associated type.

Statements

Entities that change state.

Some programming languages have no statements (*Haskell*). In some languages statements and expressions are mixed up (*C*). Different programming paradigms support different kinds of values and abstractions.

Syntactic categories in programming languages

Declarations

What we use to build abstractions: we give names to entities.

Expressions

Entities that have a value. Expressions usually have an associated type.

Statements

Entities that change state.

Some programming languages have no statements (*Haskell*). In some languages statements and expressions are mixed up (*C*). Different programming paradigms support different kinds of values and abstractions.

Syntactic categories in programming languages

Declarations

What we use to build abstractions: we give names to entities.

Expressions

Entities that have a value. Expressions usually have an associated type.

Statements

Entities that change state.

Some programming languages have no statements (*Haskell*). In some languages statements and expressions are mixed up (*C*). Different programming paradigms support different kinds of values and abstractions.

Syntactic categories in programming languages

Declarations

What we use to build abstractions: we give names to entities.

Expressions

Entities that have a value. Expressions usually have an associated type.

Statements

Entities that change state.

Some programming languages have no statements (*Haskell*). In some languages statements and expressions are mixed up (*C*). Different programming paradigms support different kinds of values and abstractions.

Syntactic categories in `minijava` - declarations

```
class A {
  int x;
  ...

  public int f(){
    int y;
    ...
  }

  ...

}
```

- classes
  - A class can be used as the type of an object.
- fields
  - A field is a global variable inside a class, stands for a place in memory.
- methods
  - A method can be used instead of a statement or an expression.
- local variables
  - Local to a method, stands for a register or a place in memory.

Syntactic categories in `minijava` - declarations

```
class A {
  int x;
  ...

  public int f(){
    int y;
    ...
  }

  ...

}
```

- classes
  - A class can be used as the type of an object.
- fields
  - A field is a global variable inside a class, stands for a place in memory.
- methods
  - A method can be used instead of a statement or an expression.
- local variables
  - Local to a method, stands for a register or a place in memory.

Syntactic categories in `minijava` - declarations

```
class A {
  int x;
  ...

  public int f(){
    int y;
    ...
  }

  ...

}
```

- classes
  - A class can be used as the type of an object.
- fields
  - A field is a global variable inside a class, stands for a place in memory.
- methods
  - A method can be used instead of a statement or an expression.
- local variables
  - Local to a method, stands for a register or a place in memory.

Syntactic categories in `minijava` - declarations

```
class A {
  int x;
  ...

  public int f(){
    int y;
    ...
  }

  ...

}
```

- classes
  - A class can be used as the type of an object.
- fields
  - A field is a global variable inside a class, stands for a place in memory.
- methods
  - A method can be used instead of a statement or an expression.
- local variables
  - Local to a method, stands for a register or a place in memory.

Syntactic categories in `minijava` - declarations

```
class A {
  int x;
  ...

  public int f(){
    int y;
    ...
  }

  ...

}
```

- classes
  - A class can be used as the type of an object.
- fields
  - A field is a global variable inside a class, stands for a place in memory.
- methods
  - A method can be used instead of a statement or an expression.
- local variables
  - Local to a method, stands for a register or a place in memory.

Syntactic categories in `minijava` - expressions

```
3 true false
x
x+98*y<x*y*y
a.f()
new A()
arr[39]
```

- constants
- identifiers
- using operators
- calling a method
- object creation
- array element

What are the values of these expressions?

What are the values of identifiers? Are the expressions meaningful?

Syntactic categories in `minijava` - expressions

```
3 true false
x
x+98*y<x*y*y
a.f()
new A()
arr[39]
```

- constants
- identifiers
- using operators
- calling a method
- object creation
- array element

What are the values of these expressions?

What are the values of identifiers? Are the expressions meaningful?

Syntactic categories in `minijava` - expressions

```
3 true false
x
x+98*y<x*y*y
a.f()
new A()
arr[39]
```

- constants
- identifiers
- using operators
- calling a method
- object creation
- array element

What are the values of these expressions?

What are the values of identifiers? Are the expressions meaningful?

Syntactic categories in `minijava` - expressions

```
3 true false
x
x+98*y<x*y*y
a.f()
new A()
arr[39]
```

- constants
- identifiers
- using operators
- calling a method
- object creation
- array element

What are the values of these expressions?

What are the values of identifiers? Are the expressions meaningful?

Syntactic categories in `minijava` - expressions

```
3 true false
x
x+98*y<x*y*y
a.f()
new A()
arr[39]
```

- constants
- identifiers
- using operators
- calling a method
- object creation
- array element

What are the values of these expressions?

What are the values of identifiers? Are the expressions meaningful?

Syntactic categories in `minijava` - expressions

```
3 true false
x
x+98*y<x*y*y
a.f()
new A()
arr[39]
```

- constants
- identifiers
- using operators
- calling a method
- object creation
- array element

What are the values of these expressions?

What are the values of identifiers? Are the expressions meaningful?

Syntactic categories in `minijava` - expressions

```
3 true false
x
x+98*y<x*y*y
a.f()
new A()
arr[39]
```

- constants
- identifiers
- using operators
- calling a method
- object creation
- array element

What are the values of these expressions?

What are the values of identifiers? Are the expressions meaningful?

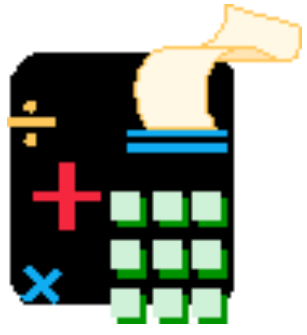
Syntactic categories in `minijava` - expressions

```
3 true false
x
x+98*y<x*y*y
a.f()
new A()
arr[39]
```

- constants
- identifiers
- using operators
- calling a method
- object creation
- array element

What are the values of these expressions?

What are the values of identifiers? Are the expressions meaningful?



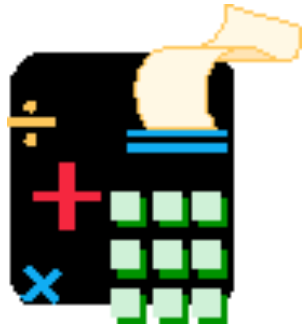
Syntactic categories in `minijava` - expressions

```
3 true false
x
x+98*y<x*y*y
a.f()
new A()
arr[39]
```

- constants
- identifiers
- using operators
- calling a method
- object creation
- array element

What are the values of these expressions?

What are the values of identifiers? Are the expressions meaningful?



Syntactic categories in `minijava` - statements

```
x = a.f();
arr[38]=x;
System.out.println(x);
if (x<y) max=y;
   else  max=x;
while(x<y)
   x=x+y;
```

assignment

array assign

output

conditional

loop

Syntactic categories in `minijava` - statements

```
x = a.f();
arr[38]=x;
System.out.println(x);
if (x<y) max=y;
   else  max=x;
while(x<y)
   x=x+y;
```

assignment

array assign

output

conditional

loop

Syntactic categories in `minijava` - statements

```
x = a.f();
arr[38]=x;
System.out.println(x);
if (x<y) max=y;
   else  max=x;
while(x<y)
   x=x+y;
```

assignment

array assign

output

conditional

loop

Syntactic categories in `minijava` - statements

```
x = a.f();
arr[38]=x;
System.out.println(x);
if (x<y) max=y;
   else  max=x;
while(x<y)
   x=x+y;
```

assignment

array assign

output

conditional

loop

Syntactic categories in `minijava` - statements

```
x = a.f();
arr[38]=x;
System.out.println(x);
if (x<y) max=y;
   else  max=x;
while(x<y)
   x=x+y;
```

assignment

array assign

output

conditional

loop

Syntactic categories in `minijava` - statements

```
x = a.f();
arr[38]=x;
System.out.println(x);
if (x<y) max=y;
   else  max=x;
while(x<y)
   x=x+y;
```

assignment

array assign

output

conditional

loop

Overview of the grammar for `minijava`

goal → *mainClass classDeclarationList*

mainClass → `CLASS ID '{' PUBLIC STATIC VOID MAIN '('`
`STRING '[' ']' ID ')' '{'` *statement* `'}'`

classDeclarationList → *classDeclarationList classDeclaration*
| ∈

classDeclaration → `CLASS ID '{'` *varDeclarationList*
metDeclarationList `'}'`

Overview of the grammar for `minijava`

goal → *mainClass classDeclarationList*

mainClass → `CLASS ID '{' PUBLIC STATIC VOID MAIN '('`
`STRING '[' ']' ID ')' '{'` *statement* `'}'`

classDeclarationList → *classDeclarationList classDeclaration*
| ∈

classDeclaration → `CLASS ID '{'` *varDeclarationList*
metDeclarationList `'}'`

Overview of the grammar for `minijava`

goal → *mainClass classDeclarationList*

mainClass → `CLASS ID '{' PUBLIC STATIC VOID MAIN '('`
`STRING '[' ']' ID ')' '{'` *statement* `'}'`

classDeclarationList → *classDeclarationList classDeclaration*
| ∈

classDeclaration → `CLASS ID '{'` *varDeclarationList*
metDeclarationList `'}'`

Overview of the grammar for `minijava`

goal → *mainClass classDeclarationList*

mainClass → `CLASS ID '{' PUBLIC STATIC VOID MAIN '('`
`STRING '[' ']' ID ')' '{'` *statement* `'}'`

classDeclarationList → *classDeclarationList classDeclaration*
| ∈

classDeclaration → `CLASS ID '{'` *varDeclarationList*
metDeclarationList `'}'`

Overview of the grammar

metDeclaration → PUBLIC *type* ID '(' *parameters* ')' '{'
varDeclarationList statementList
RETURN *exp* '}'

statement → '{' *statementList* '}'
| IF '(' *exp* ')' *statement* ELSE *statement*
| WHILE '(' *exp* ')' *statement*
| SYSTEMOUT '(' *exp* ')' ';'
| ID '=' *exp* ';'
| ID '[' *exp* ']' '=' *exp* ';'

Overview of the grammar

metDeclaration → PUBLIC *type* ID '(' *parameters* ')' '{'
varDeclarationList statementList
RETURN *exp* '}'

statement → '{' *statementList* '}'
| IF '(' *exp* ')' *statement* ELSE *statement*
| WHILE '(' *exp* ')' *statement*
| SYSTEMOUT '(' *exp* ')' ';'
| ID '=' *exp* ';'
| ID '[' *exp* ']' '=' *exp* ';'

Overview of the grammar

metDeclaration → PUBLIC *type* ID '(' *parameters* ')' '{'
varDeclarationList statementList
RETURN *exp* '}'

statement → '{' *statementList* '}'
| IF '(' *exp* ')' *statement* ELSE *statement*
| WHILE '(' *exp* ')' *statement*
| SYSTEMOUT '(' *exp* ')' ';'
| ID '=' *exp* ';'
| ID '[' *exp* ']' '=' *exp* ';'

Overview of the grammar

metDeclaration → PUBLIC *type* ID '(' *parameters* ')' '{'
varDeclarationList statementList
RETURN *exp* '}'

statement → '{' *statementList* '}'
| IF '(' *exp* ')' *statement* ELSE *statement*
| WHILE '(' *exp* ')' *statement*
| SYSTEMOUT '(' *exp* ')' ';'
| ID '=' *exp* ';'
| ID '[' *exp* ']' '=' *exp* ';'

Overview of the grammar

metDeclaration → PUBLIC *type* ID '(' *parameters* ')' '{'
varDeclarationList statementList
RETURN *exp* '}'

statement → '{' *statementList* '}'
| IF '(' *exp* ')' *statement* ELSE *statement*
| WHILE '(' *exp* ')' *statement*
| SYSTEMOUT '(' *exp* ')' ';'
| ID '=' *exp* ';'
| ID '[' *exp* ']' '=' *exp* ';'

Overview of the grammar

metDeclaration → PUBLIC *type* ID '(' *parameters* ')' '{'
varDeclarationList statementList
RETURN *exp* '}'

statement → '{' *statementList* '}'
| IF '(' *exp* ')' *statement* ELSE *statement*
| WHILE '(' *exp* ')' *statement*
| SYSTEMOUT '(' *exp* ')' ';'
| ID '=' *exp* ';'
| ID '[' *exp* ']' '=' *exp* ';'

Overview of the grammar

metDeclaration → PUBLIC *type* ID '(' *parameters* ')' '{'
varDeclarationList statementList
RETURN *exp* '}'

statement → '{' *statementList* '}'
| IF '(' *exp* ')' *statement* ELSE *statement*
| WHILE '(' *exp* ')' *statement*
| SYSTEMOUT '(' *exp* ')' ';'
| ID '=' *exp* ';'
| ID '[' *exp* ']' '=' *exp* ';'

Overview of the grammar

metDeclaration → PUBLIC *type* ID '(' *parameters* ')' '{'
varDeclarationList statementList
RETURN *exp* '}'

statement → '{' *statementList* '}'
| IF '(' *exp* ')' *statement* ELSE *statement*
| WHILE '(' *exp* ')' *statement*
| SYSTEMOUT '(' *exp* ')' ';'
| ID '=' *exp* ';'
| ID '[' *exp* ']' '=' *exp* ';'

Remarks

The grammar in the appendix of the book describes a small subset of Java called minijava. Do not add things to it! We will compile minijava!

When understanding the grammar, start from the goal. But when implementing it in jacc start with expressions, then statements and then the declarations!

Read in the jacc manual how to do lists, lists with separators, etc.

The grammar will have some conflicts. Test it a lot in order to decide whether you can let jacc solve them or whether you will have to modify the grammar!

Remarks

The grammar in the appendix of the book describes a small subset of Java called minijava. Do not add things to it! We will compile minijava!

When understanding the grammar, start from the goal. But when implementing it in jacc start with expressions, then statements and then the declarations!

Read in the jacc manual how to do lists, lists with separators, etc.

The grammar will have some conflicts. Test it a lot in order to decide whether you can let jacc solve them or whether you will have to modify the grammar!

Remarks

The grammar in the appendix of the book describes a small subset of Java called minijava. Do not add things to it! We will compile minijava!

When understanding the grammar, start from the goal. But when implementing it in `jacc` start with expressions, then statements and then the declarations!

Read in the `jacc` manual how to do lists, lists with separators, etc.

The grammar will have some conflicts. Test it a lot in order to decide whether you can let `jacc` solve them or whether you will have to modify the grammar!

Remarks

The grammar in the appendix of the book describes a small subset of Java called minijava. Do not add things to it! We will compile minijava!

When understanding the grammar, start from the goal. But when implementing it in `jacc` start with expressions, then statements and then the declarations!

Read in the `jacc` manual how to do lists, lists with separators, etc.

The grammar will have some conflicts. Test it a lot in order to decide whether you can let `jacc` solve them or whether you will have to modify the grammar!

Remarks

The grammar in the appendix of the book describes a small subset of Java called minijava. Do not add things to it! We will compile minijava!

When understanding the grammar, start from the goal. But when implementing it in `jacc` start with expressions, then statements and then the declarations!

Read in the `jacc` manual how to do lists, lists with separators, etc.

The grammar will have some conflicts. Test it a lot in order to decide whether you can let `jacc` solve them or whether you will have to modify the grammar!

Storing the structure

While the parser recognizes a program, we cannot calculate a value but we can calculate a representation of the structure!

We forget about connecting terminals and about the derivation and concentrate in the other syntactic categories that are used!

Storing the structure

While the parser recognizes a program, we cannot calculate a value but we can calculate a representation of the structure!

We forget about connecting terminals and about the derivation and concentrate in the other syntactic categories that are used!

What do we need?

Main class

We only need the name and the statement!

Class Declaration

We only need the name, the list of variable declarations and the list of method declarations.

Method declaration

We only need the result type, the name, the list of parameters, the list of variable declarations, the statement list and the returned expression.

For a statement, it depends . . .

What do we need?

Main class

We only need the name and the statement!

Class Declaration

We only need the name, the list of variable declarations and the list of method declarations.

Method declaration

We only need the result type, the name, the list of parameters, the list of variable declarations, the statement list and the returned expression.

For a statement, it depends . . .

What do we need?

Main class

We only need the name and the statement!

Class Declaration

We only need the name, the list of variable declarations and the list of method declarations.

Method declaration

We only need the result type, the name, the list of parameters, the list of variable declarations, the statement list and the returned expression.

For a statement, it depends . . .

What do we need?

Main class

We only need the name and the statement!

Class Declaration

We only need the name, the list of variable declarations and the list of method declarations.

Method declaration

We only need the result type, the name, the list of parameters, the list of variable declarations, the statement list and the returned expression.

For a statement, it depends . . .

What do we need?

Main class

We only need the name and the statement!

Class Declaration

We only need the name, the list of variable declarations and the list of method declarations.

Method declaration

We only need the result type, the name, the list of parameters, the list of variable declarations, the statement list and the returned expression.

For a statement, it depends . . .

Statements

Assign

We need the identifier and the expression.

Array element assign

We need the identifier, the indexing expression and the right hand side expression.

If

We need the expression and two statements!

While

We need and expression and a statement!

Statements

Assign

We need the identifier and the expression.

Array element assign

We need the identifier, the indexing expression and the right hand side expression.

If

We need the expression and two statements!

While

We need and expression and a statement!

Statements

Assign

We need the identifier and the expression.

Array element assign

We need the identifier, the indexing expression and the right hand side expression.

If

We need the expression and two statements!

While

We need and expression and a statement!

Statements

Assign

We need the identifier and the expression.

Array element assign

We need the identifier, the indexing expression and the right hand side expression.

If

We need the expression and two statements!

While

We need and expression and a statement!

Statements

Assign

We need the identifier and the expression.

Array element assign

We need the identifier, the indexing expression and the right hand side expression.

If

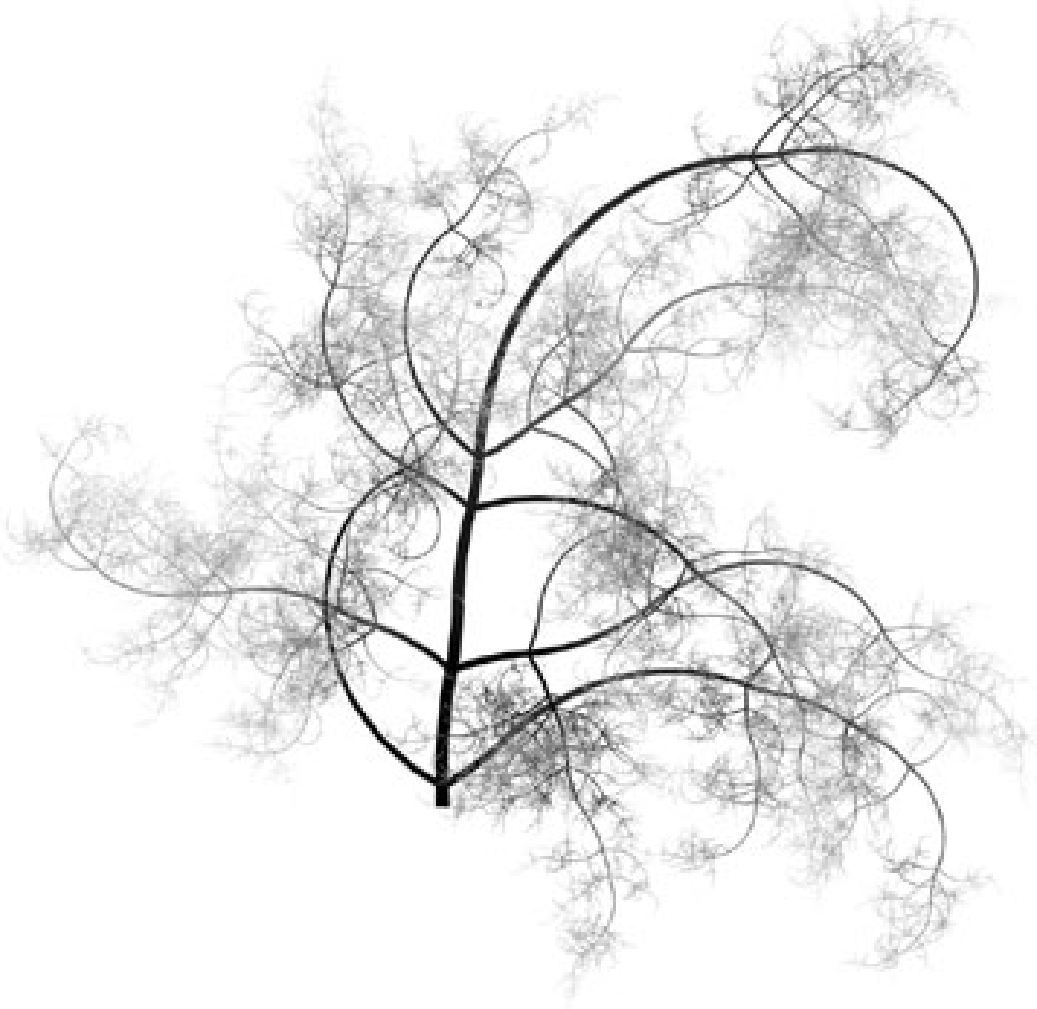
We need the expression and two statements!

While

We need and expression and a statement!

Trees

All this looks very much like trees with diferent type and number of children for different types of nodes!



Abstract syntax in Java

We need a datastructure in Java for the abstract syntax trees.

Problem

There will be a class for each type of node. But we want that all types of statement nodes can be used where a statement is needed! (and the same for expressions by the way.)

Containers and components

Bricks are used as components, contained in a construction. But the construction can be used as a component in a larger construction!

Abstract syntax in Java

We need a datastructure in Java for the abstract syntax trees.

Problem

There will be a class for each type of node. But we want that all types of statement nodes can be used where a statement is needed! (and the same for expressions by the way.)

Containers and components

Bricks are used as components, contained in a construction. But the construction can be used as a component in a larger construction!

Abstract syntax in Java

We need a datastructure in Java for the abstract syntax trees.

Problem

There will be a class for each type of node. But we want that all types of statement nodes can be used where a statement is needed! (and the same for expressions by the way.)

Containers and components

Bricks are used as components, contained in a construction. But the construction can be used as a component in a larger construction!

Statements

Node for IF

```
public class If {
  public Exp e;
  public Statement s1,s2;

  public If(Exp ae, Statement as1, Statement as2) {
    e=ae; s1=as1; s2=as2;
  }
}
```

How do we see that this builds a statement, that can be used where a statement is required?

Statements

Node for IF

```
public class If {
  public Exp e;
  public Statement s1,s2;

  public If(Exp ae, Statement as1, Statement as2) {
    e=ae; s1=as1; s2=as2;
  }
}
```

How do we see that this builds a statement, that can be used where a statement is required?

Statements

Node for IF

```
public class If extends Statement{
  public Exp e;
  public Statement s1,s2;

  public If(Exp ae, Statement as1, Statement as2) {
    e=ae; s1=as1; s2=as2;
  }
}
```

Statements

```
public abstract class Statement {}
```

Statements

Node for IF

```
public class If extends Statement{
  public Exp e;
  public Statement s1,s2;

  public If(Exp ae, Statement as1, Statement as2) {
    e=ae; s1=as1; s2=as2;
  }
}
```

Statements

```
public abstract class Statement {}
```

Building syntax trees

As the parser recognizes sentences of the different forms, it will build the abstract syntax trees.

Example

```
statement   :  IF '(' exp ')' statement ELSE statement
            { $$ = new If($3, $5, $7); }
```

There is an abstract class for the syntactic category *statement* and one class inheriting from it for each production for *statement*!

And the same idea is used for expressions!

Building syntax trees

As the parser recognizes sentences of the different forms, it will build the abstract syntax trees.

Example

```
statement    :   IF '(' exp ')' statement ELSE statement
                 { $$ = new If($3, $5, $7); }
```

There is an abstract class for the syntactic category *statement* and one class inheriting from it for each production for *statement*!

And the same idea is used for expressions!

Building syntax trees

As the parser recognizes sentences of the different forms, it will build the abstract syntax trees.

Example

```
statement    :   IF '(' exp ')' statement ELSE statement
                 { $$ = new If($3, $5, $7); }
```

There is an abstract class for the syntactic category *statement* and one class inheriting from it for each production for *statement*!

And the same idea is used for expressions!

Building syntax trees

As the parser recognizes sentences of the different forms, it will build the abstract syntax trees.

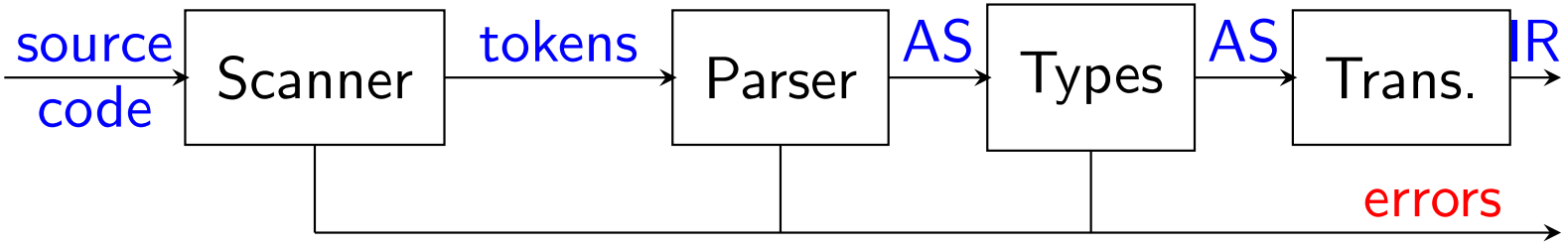
Example

```
statement    :   IF '(' exp ')' statement ELSE statement
                 { $$ = new If($3, $5, $7); }
```

There is an abstract class for the syntactic category *statement* and one class inheriting from it for each production for *statement*!

And the same idea is used for expressions!

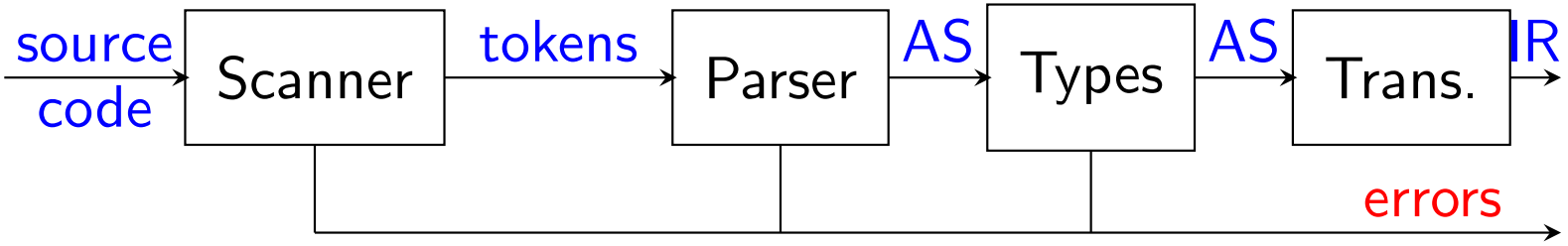
Parsing `minijava`



The project in the course continues by using `jacc` to build a parser for `minijava` that, in the process of parsing a `minijava` program, creates an abstract syntax tree for the program!

You get the datastructure for the abstract syntax! (many classes, described in chapter 4 of the course book).

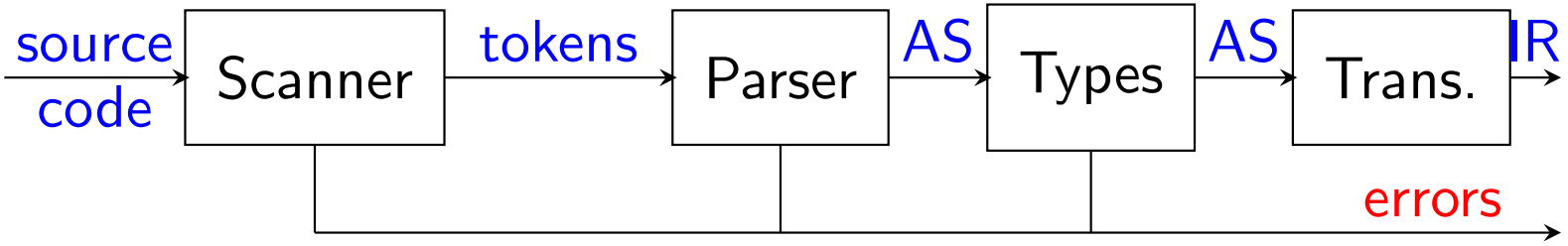
Parsing `minijava`



The project in the course continues by using `jacc` to build a parser for `minijava` that, in the process of parsing a `minijava` program, creates an abstract syntax tree for the program!

You get the datastructure for the abstract syntax! (many classes, described in chapter 4 of the course book).

Parsing `minijava`



The project in the course continues by using `jacc` to build a parser for `minijava` that, in the process of parsing a `minijava` program, creates an abstract syntax tree for the program!

You get the datastructure for the abstract syntax! (many classes, described in chapter 4 of the course book).