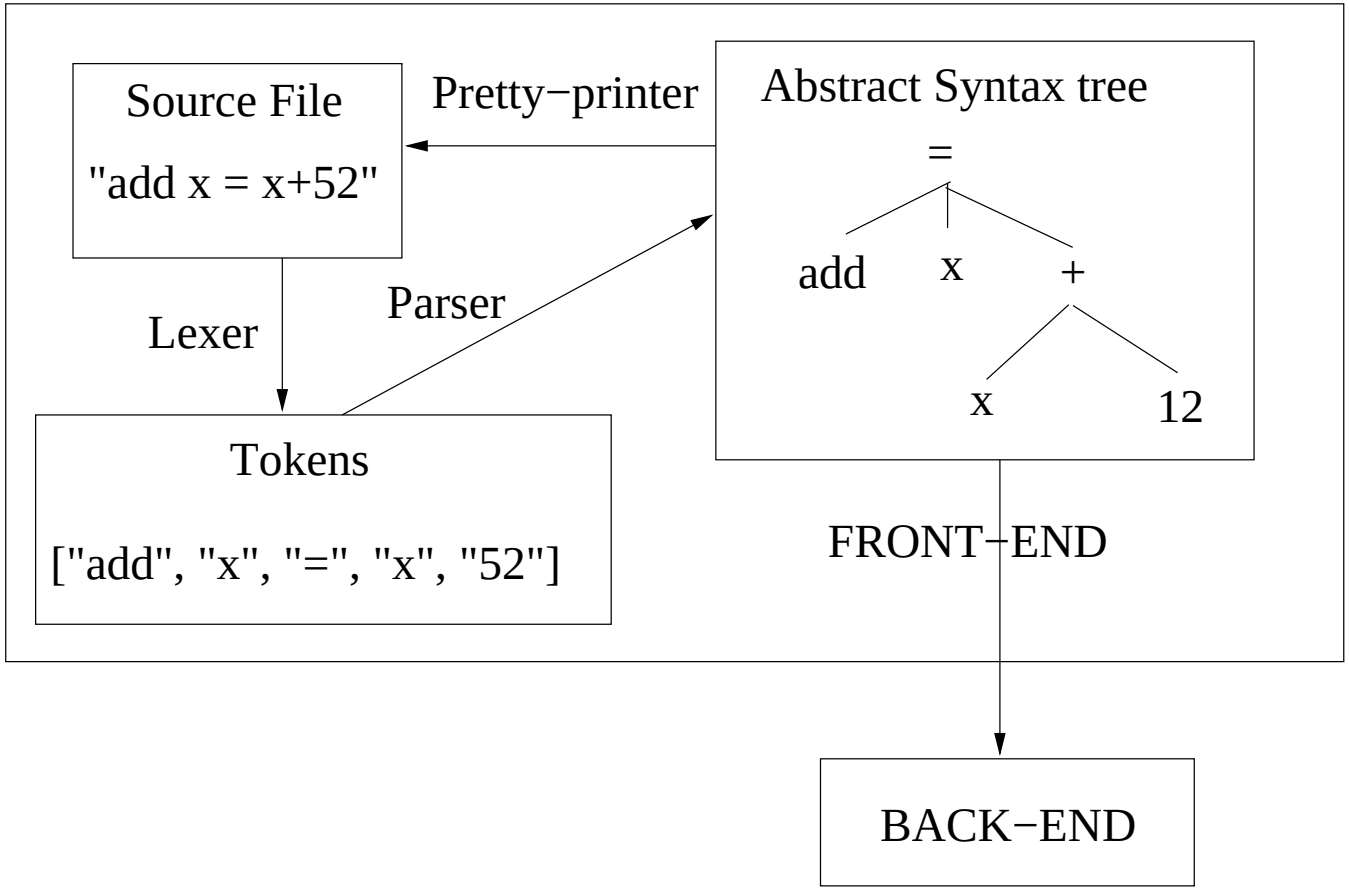
BNF Converter
Guest Lecture

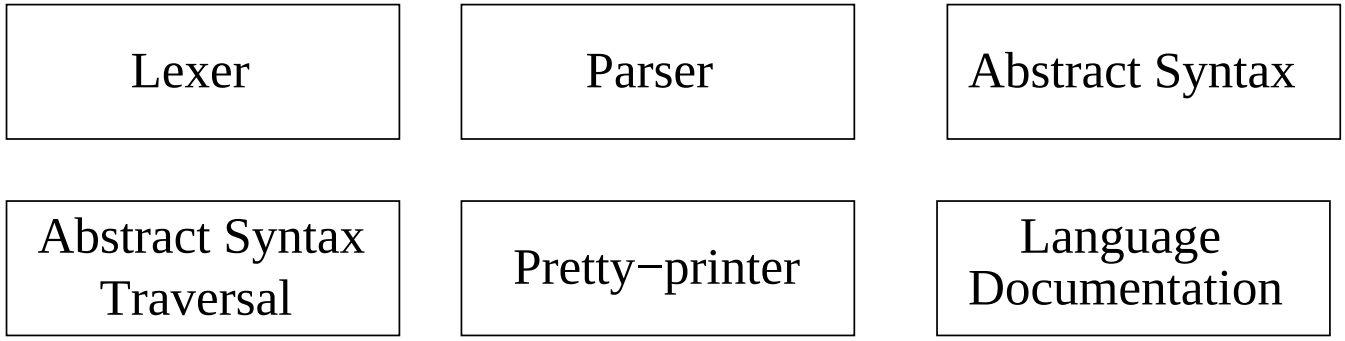
Björn Bringert
bringert@cs.chalmers.se
2007
Many slides shamelessly stolen from
Markus Forsberg

BNF Converter
What is a Compiler Front-End?



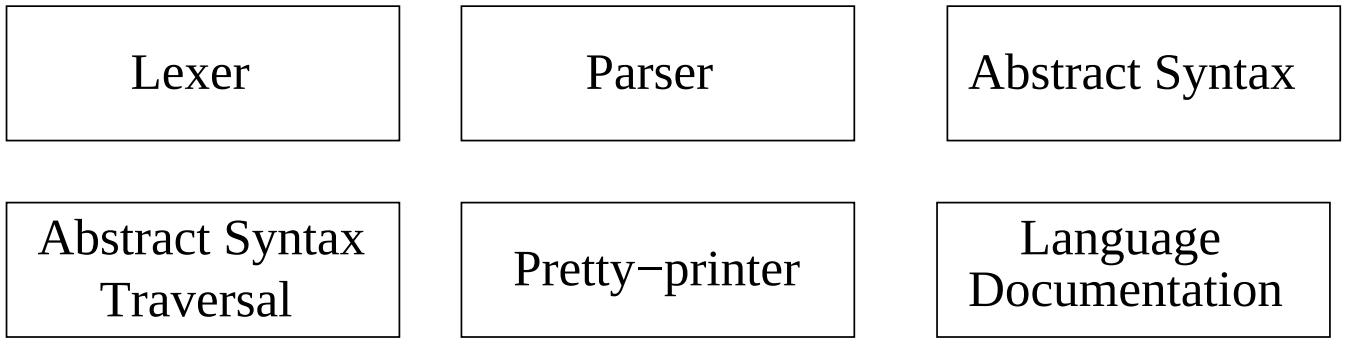
A Compiler Front-End for a Formal Language

- We start with an Idea about the formal language:
  - Language Specification
  - In the mind of the implementer
- We continue by developing a set of modules, usually with the help of existing tools.



Problem: Consistency

- Hard to keep all modules consistent!
- Say that we want to extend our language with a new language construct. Then we have to change every module!



Problem: Boring Code

- We have to write a lot of boring code.

Example: Happy parser generator code.

```
...
Stm :: { Stm }
Stm :
    Labeled_stm           { LabelS $1 }
  | Compound_stm          { CompS $1 }
  | Expression_stm        { ExprS $1 }
  | Selection_stm         { SelS $1 }
  | Iter_stm              { IterS $1 }
  | Jump_stm              { JumpS $1 }
...
```

Problem: Stuck in a particular Programming Language

- We end up with a compiler front-end in a specific programming language.
- But we (may) want to design in a declarative language.
- and as a final product use an imperative language (e.g. a compiler in C).
- or incorporate our language in a system written in another language
- Then we have to rewrite everything! Irggghhh....

Solution

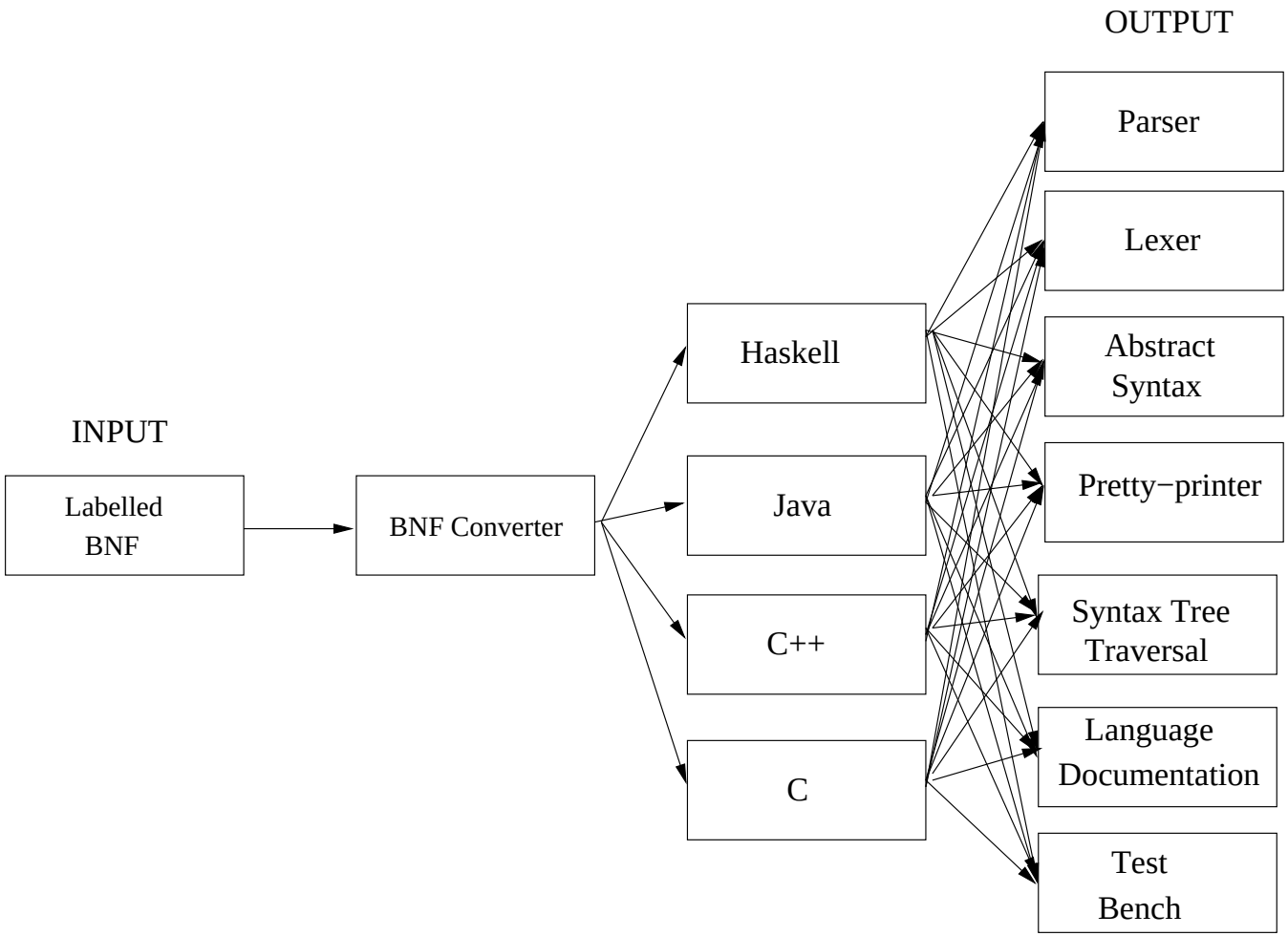
- Use a single source to generate all modules.
- Use a simple formalism for the single source.
- Use a declarative style for the single source.
  Describe instead of implement the language.

Example (Labelled BNF, LBNF):

```
Fact . Clause ::= Predicate ;
Rule . Clause ::= Predicate ":-" [Predicate] ;

token UIdent (upper (letter | digit | '_')*) ;
```

System Overview: BNF Converter



Example Language

- Expression language with integers, doubles and addition.
- Syntax in LBNF (in the file `demo.cf`):

```
entrypoints Exp;

EAdd. Exp  ::= Exp "+" Exp2 ;
_.    Exp  ::= Exp2 ;
EInt. Exp2 ::= Integer ;
EDbl. Exp2 ::= Double ;
```

Haskell Backend

- Generate Haskell code with `bnfc -m -haskell demo.cf`
- Categories become algebraic datatypes.
- Labels become data constructors.
- Abstract syntax of example grammar:

```
data Exp =
   EAdd Exp Exp
 | EInt Integer
 | EDbl Double
```

Type Checker in Haskell

- Typing rules:
  - Integer literals have type Int.
  - Double literals have type Double.
  - $E_1 + E_2$ has the same type as E_1 and E_2.
- Haskell code:

```
inferType :: Exp -> Err Type
inferType (EAdd e1 e2) =
    do t1 <- inferType e1
       t2 <- inferType e2
       if t1 == t2 then return t1 else fail "add"
inferType (EInt _) = return TInt
inferType (EDbl _) = return TDbl
```

Haskell Lexer and Parser

- Generated lexer:

```
myLexer :: String -> [Token]
```

- Generated parser:

```
pExp :: [Token] -> Err Exp
```

- Lexing, parsing and type inference:

```
checkProg :: String -> Err Type
checkProg s = pExp (myLexer s) >>= inferType
```

Java 1.5 Backend

- Generate Java 1.5 code with `bnfc -m -java1.5 demo.cf`
- Categories become abstract classes.
- Labels become concrete classes.
- Uses the Visitor Design pattern (a poor language's pattern matching).

Java 1.5 Abstract Syntax

```
public abstract class Exp implements java.io.Serializable {
  public abstract <R,A> R accept(Exp.Visitor<R,A> v, A arg);
  public interface Visitor <R,A> {
    public R visit(Example.Absyn.EAdd p, A arg);
    public R visit(Example.Absyn.EInt p, A arg);
    public R visit(Example.Absyn.EDbl p, A arg);
  }
}
public class EAdd extends Exp {
  public final Exp exp_1, exp_2;
  public EAdd(Exp p1, Exp p2) { exp_1 = p1; exp_2 = p2; }
  public <R,A> R accept(Exp.Visitor<R,A> v, A arg) {
    return v.visit(this, arg); }
  // implementations of equals() and hashCode()
}
```

Type Checker in Java

```
public Type inferType(Exp e) {
  return e.accept(new TypeInferrer(), null);
}
private class TypeInferrer implements Exp.Visitor<Type,Object> {
  public Type visit(EAdd p, Object arg) {
    Type t1 = inferType(p.exp_1);
    Type t2 = inferType(p.exp_2);
    if (t1.equals(t2))
      return t1;
    else
      throw new TypeException();
  }
  public Type visit(EInt p, Object arg) { return new TInt(); }
  public Type visit(EDbl p, Object arg) { return new TDbl(); }
}
```

Bonus Slide: Sequences in BNFC

- Grammar:

```
ESum. Exp ::= "sum" "(" [Exp] ")" ;
separator nonempty Exp "," ;
```

- Haskell abstract syntax:

```
data Exp = ESum [Exp] | ...
```

- Java 1.5 abstract syntax uses `List<Exp>`.
- `terminator`: a token after each element.
- Custom sequences: use `[]` and `(:)`, and optionally `(:[])`.

Bonus Slide: Precedence and Associativity 1

- `+` and `*` left-associative, `*` has higher precedence:

```
entrypoints Exp;
EAdd. Exp  ::= Exp "+" Exp2 ;
EMul. Exp2 ::= Exp2 "*" Exp3 ;
EInt. Exp3 ::= Integer ;
coercions Exp 3 ;
```

- `coercions Exp 3` generates:

```
_. Exp  ::= Exp2 ;
_. Exp2 ::= Exp3 ;
_. Exp3 ::= "(" Exp ")" ;
```

Bonus Slide: Precedence and Associativity 2

- Does not affect the abstract syntax:

```
data Exp = EAdd Exp Exp | EMul Exp Exp | EInt Integer
```

- Examples:

```
 1  +  2  * 3 → EAdd (EInt 1) (EMul (EInt 2) (EInt 3))
(1  +  2) * 3 → EMul (EAdd (EInt 1) (EInt 2)) (EInt 3)
 1  +  2  + 3 → EAdd (EAdd (EInt 1) (EInt 2)) (EInt 3)
```

Other Features and Conclusions

- BNFC has support for comments, sequences (bonus slide!), precedence, lexical rules, layout syntax, etc.
- It's easy and fast to use, $\approx$ 10 min for a simple new language.
- It's addictive: I use 4 or 5 different BNFC grammars in a single system.

The End

- Example code is available from the course homepage.
- BNFC documentation:
  `http://www.cs.chalmers.se/~markus/BNFC/`
- LBNF report: The best reference on BNFC:
  `http://www.cs.chalmers.se/~markus/BNFC/LBNF-report.pdf`
- Questions?