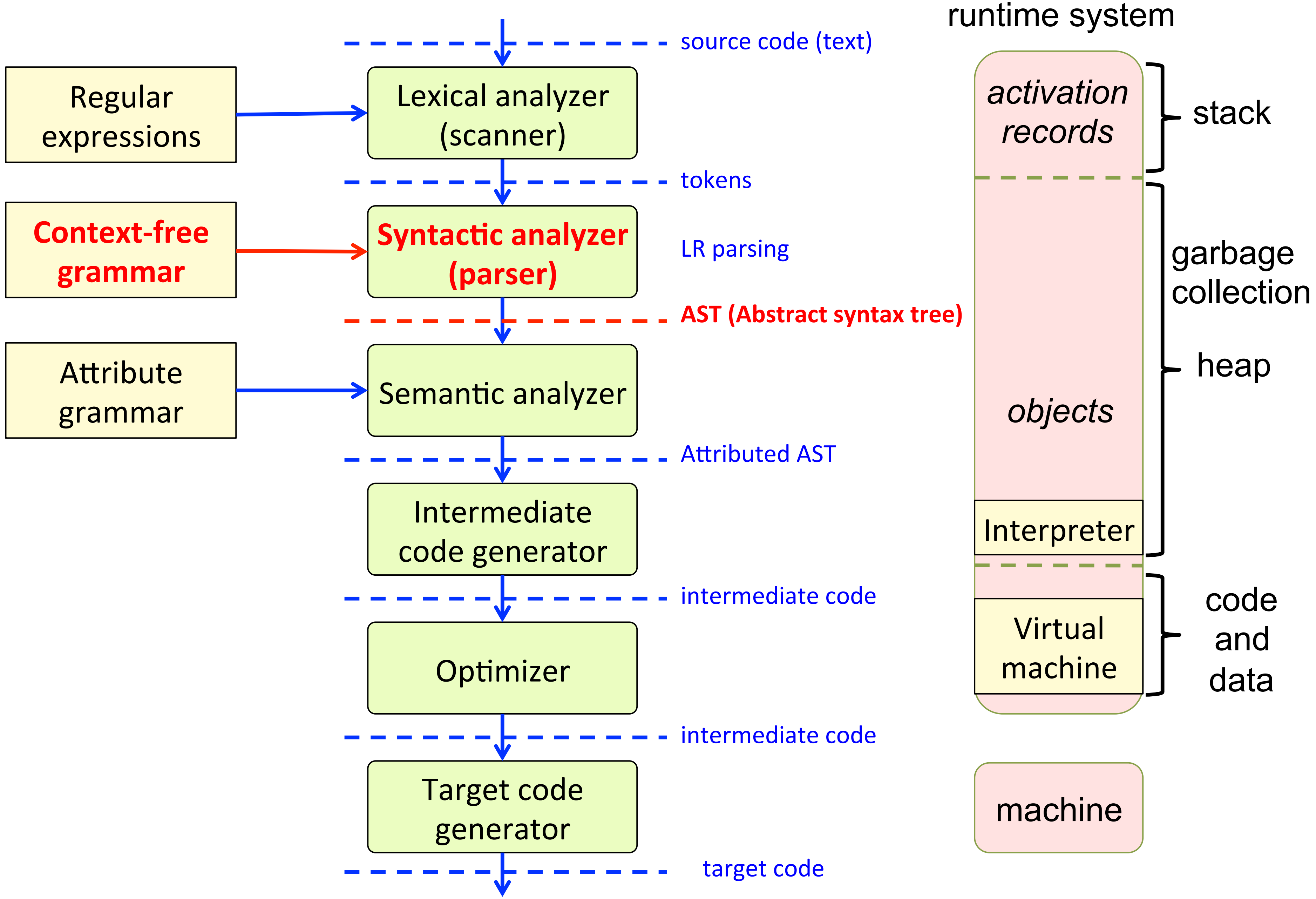
EDAN65: Compilers, Lecture 05 B

Abstract grammars

Görel Hedin

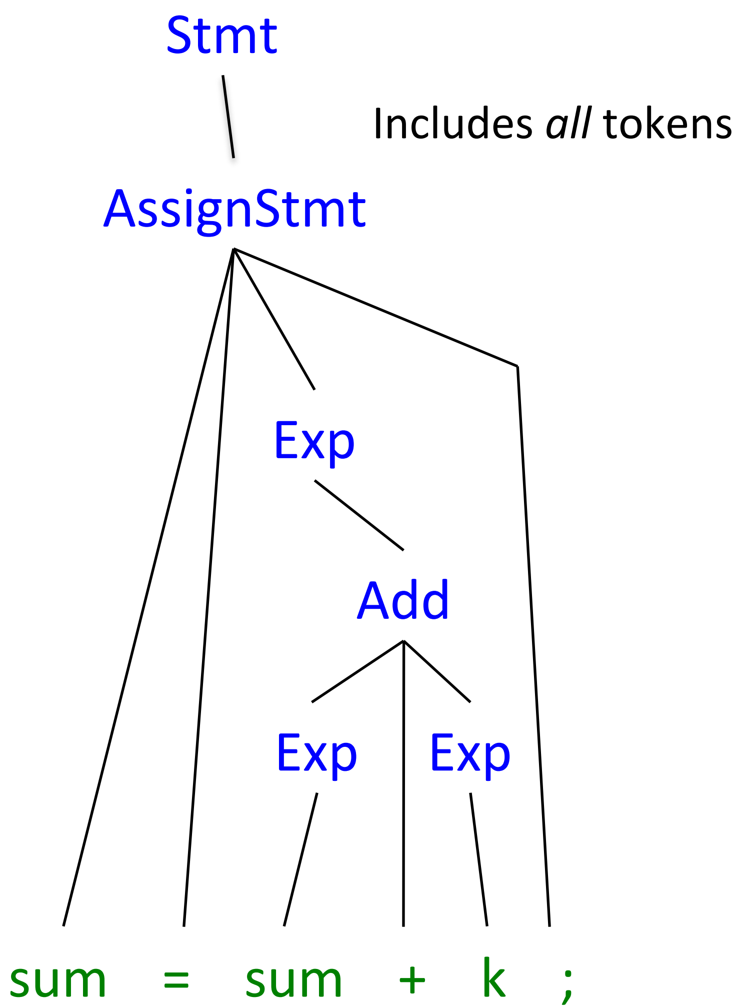
Revised: 2016-09-12

This lecture

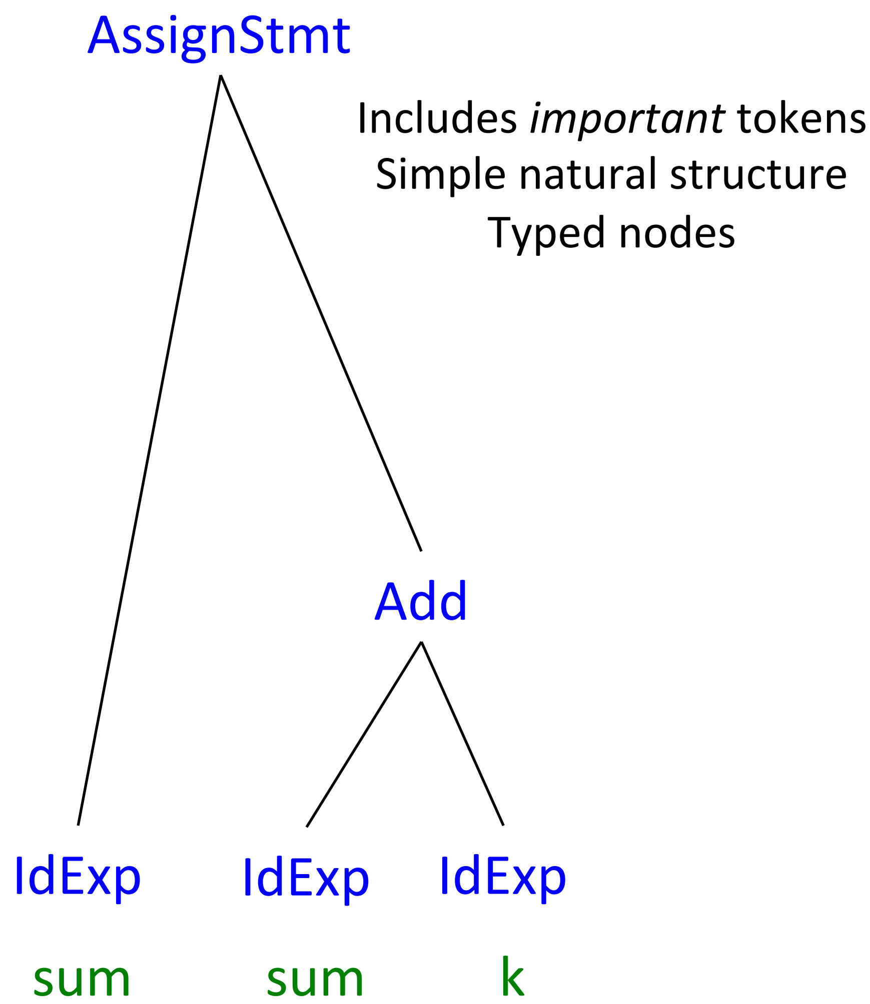


Abstract grammars

Parse tree



Abstract tree



Example: Concrete vs Abstract

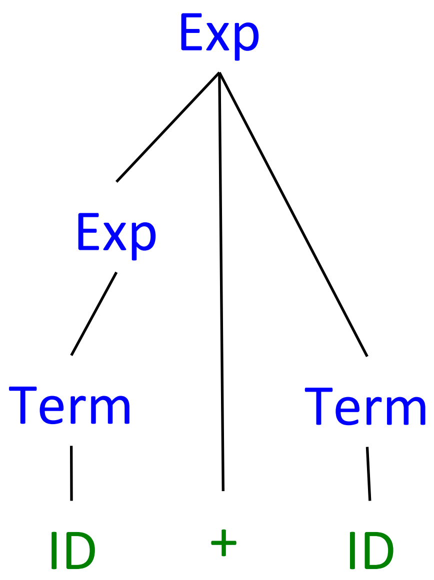
Concrete grammar

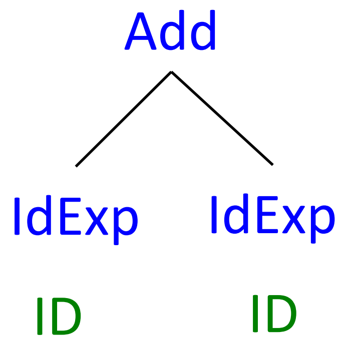
```
Exp -> Exp "+" Term
Exp -> Term
Term -> ID
```

Abstract grammar

```
Add: Exp -> Exp Exp
IdExp: Exp -> ID
```

Productions are named!





Note! Term, Factor, are needed to make the concrete grammar unambiguous.

An abstract grammar cannot be ambiguous. Term and Factor are irrelevant here.

Concrete vs Abstract grammar

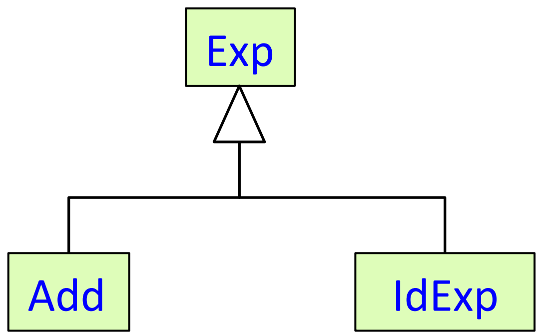
	Concrete Grammar	**Abstract Grammar**
What does it describe?	Describes the concrete text representation of programs	Describes the abstract structure of programs
Main use	Parsing text to trees	Model representing the program inside compiler.
Underlying formalism	Context-free grammar	Recursive data types
What is named?	Only nonterminals (productions are usually anonymous)	Both nonterminals and productions.
What tokens occur in the grammar?	all tokens corresponding to "words" in the text	usually only tokens with values (identifiers, literals)
	Independent of abstract structure	Independent of parser and parser algorithm

Abstract grammar vs. OO model

Abstract grammar	OO model	Other terminology used (algebraic datatypes)
nonterminal	superclass	type, sort
production	subclass	constructor, operator

Abstract grammar

```
Add: Exp -> Exp Exp
IdExp: Exp -> ID
```

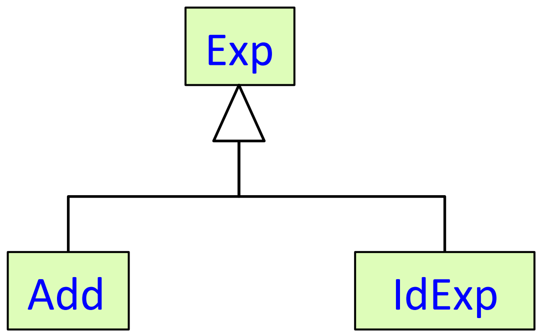


A canonical abstract grammar corresponds to a two-level class hierarchy!

Example Java implementation

Abstract grammar

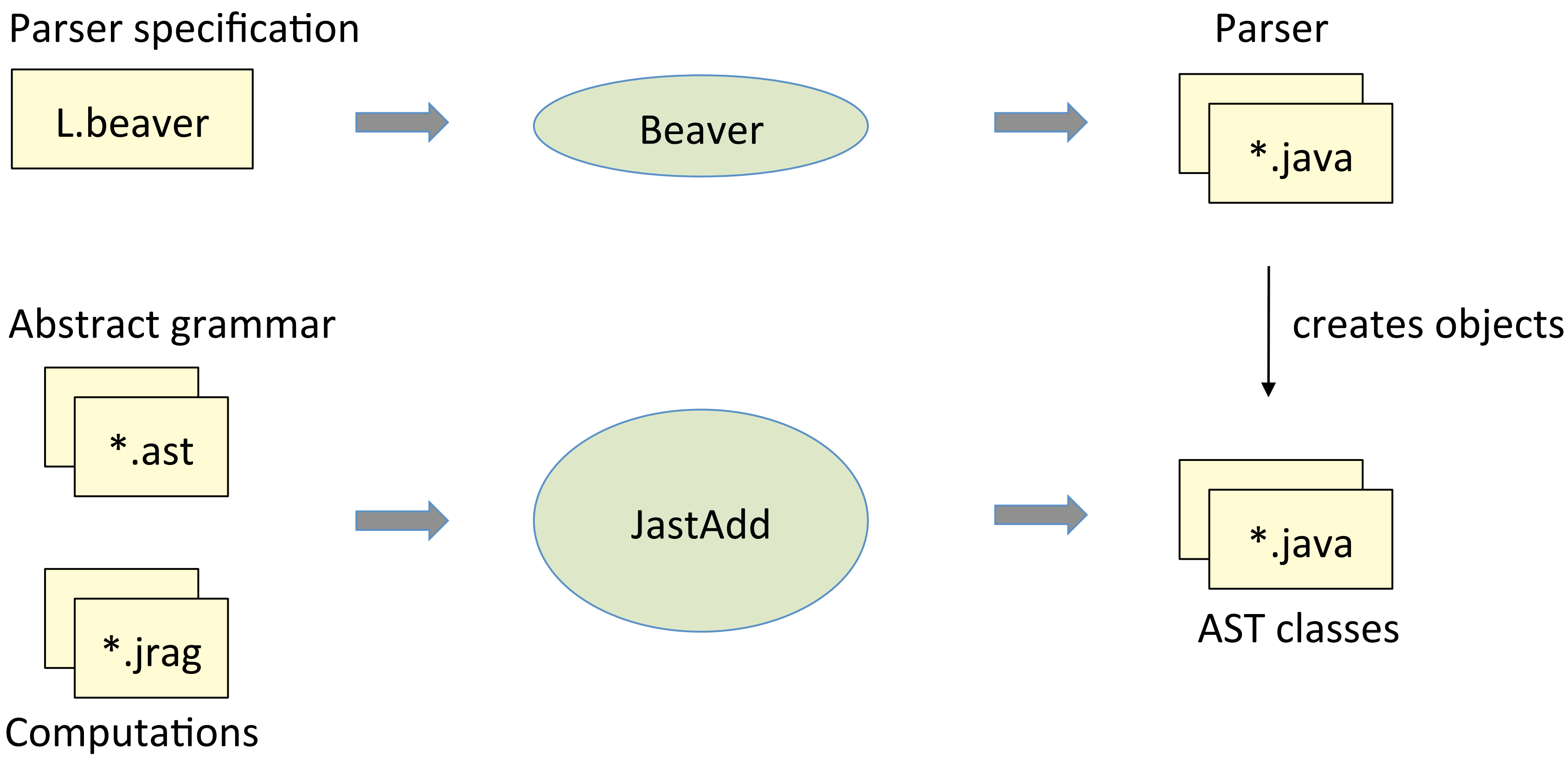
```
Add: Exp -> Exp Exp
IdExp: Exp -> ID
```




```java
abstract class Exp {
}
class Add extends Exp {
  Exp exp1, exp2;
}
class IdExp extends Exp {
  String ID;
}
```

JastAdd

- A compiler generation tool. Generates Java code.
- Supports ASTs and modular computations on ASTs.
- JastAdd: "Just add computations to the ast"
- Independent of the parser used.
- Developed at LTH, see http://jastadd.org



JastAdd abstract grammars
(compared to canonical abstract grammars)

```
Program ::= Stmt*;
abstract Stmt;
Assignment : Stmt ::= IdExpr Expr;
IfStmt : Stmt ::= Expr Then:Stmt [Else:Stmt];
abstract Expr;
IdExpr : Expr ::= <ID:String>;
IntExpr : Expr ::= <INT:String>;
BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr;
```

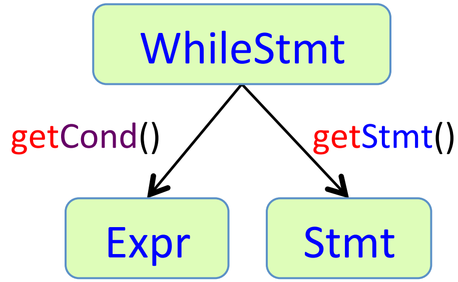
- Classes instead of nonterminals and productions
- Classes can be abstract (like in Java)
- Arbitrarily deep inheritance hierarchy (not just two levels)
- Support for *optional*, *list*, and *token* components
- Components can be named
- Right-hand side can be inherited from superclass (see BinExpr).
- No parentheses! You need to name all node classes in the AST.

Generated Java API, ordinary components

```
abstract Stmt;
WhileStmt : Stmt ::= Cond:Expr Stmt;
```

```
abstract class Stmt extends ASTNode {}

class WhileStmt extends Stmt {
  Expr getCond();
  Stmt getStmt();
}
```



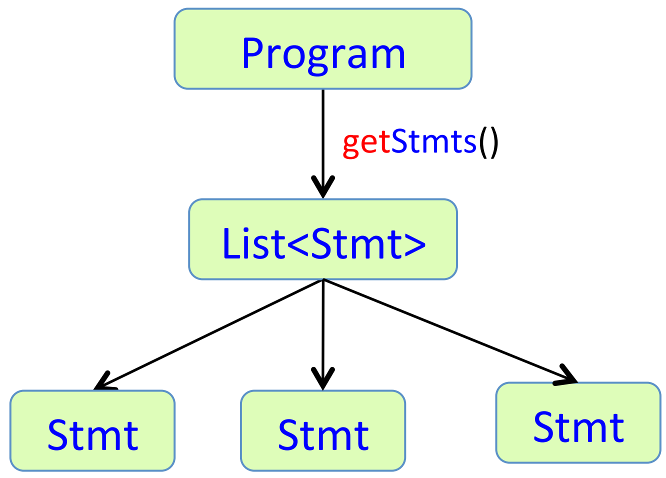
Example AST

- A general class ASTNode is used as implicit superclass.
- A traversal API with *get* methods is generated.
- If component names are given, they are used in the API (getCond).
- Otherwise the type names are used (getStmt).

Generated Java API, lists

```
Program ::= Stmt*;
```

```
class Program extends ASTNode {
  int getNumStmt();   // 0 if empty
  Stmt getStmt(int i); // numbered from 0
  List<Stmt> getStmts(); // iterator
}
```



Example AST

The list is represented by a List object that can be used as an iterator:

```
Program p = ...;
for (Stmt s : p.getStmts()) {
  ...
}
```

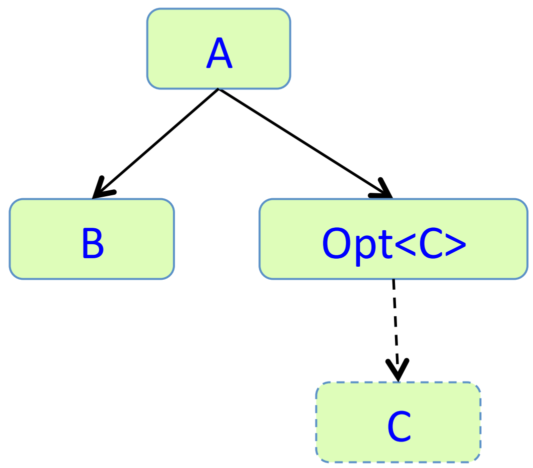
Or access a specific statement:

```
Program p = ...;
if (p.getNumStmt() >= 1) {
  Stmt s = p.getStmt(0);
  ...
}
```

Generated Java API, optionals

```
A ::= B [C];
```

```
class A extends ASTNode {
  B getB();
  boolean hasC();
  C getC();       //Exception if not hasC()
}
```



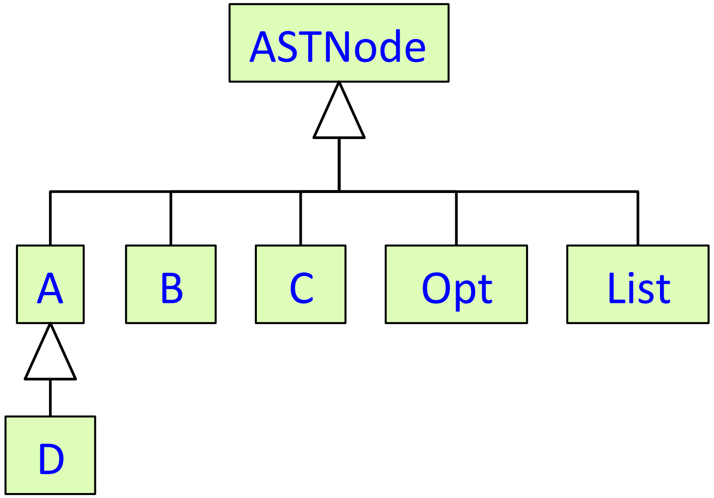
Example AST

- The traversal API includes a *has* method for the optional component.

General traversal

Abstract grammar

```
A ::= B [C];
B ::= ...;
C ::= ...;
D : A ::= ...;
```



Will stop also at Opt and List nodes.
Can be used for general traversal of the children of a node.

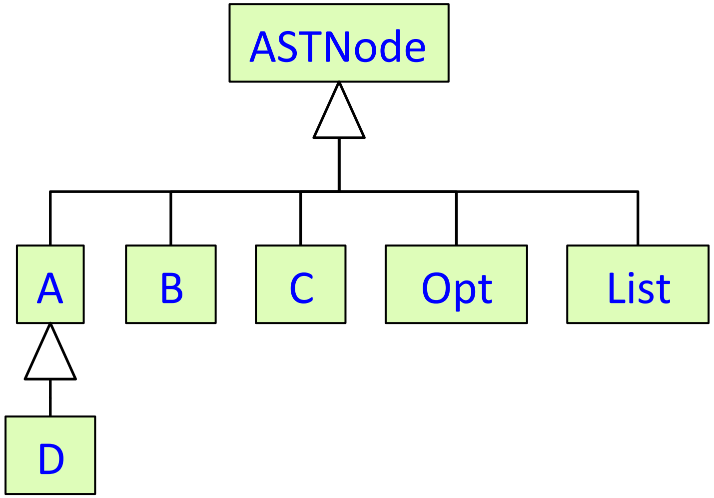
```
class ASTNode {
  Iterable astChildren(); //Iterator for the children
}
```

```
ASTNode n = ...;
for (ASTNode child : n.astChildren()) {
  ...
}
```

Low-level traversal API

Abstract grammar

```
A ::= B [C];
B ::= ...;
C ::= ...;
D : A ::= ...;
```



Will stop also at Opt and List nodes.
Not recommended. Use iterator or high-level API instead – much more readable.

```
class ASTNode {
  int getNumChild();
  ASTNode getChild(int i);
  ASTNode getParent(); // null for the root
}
```

Connection to Beaver

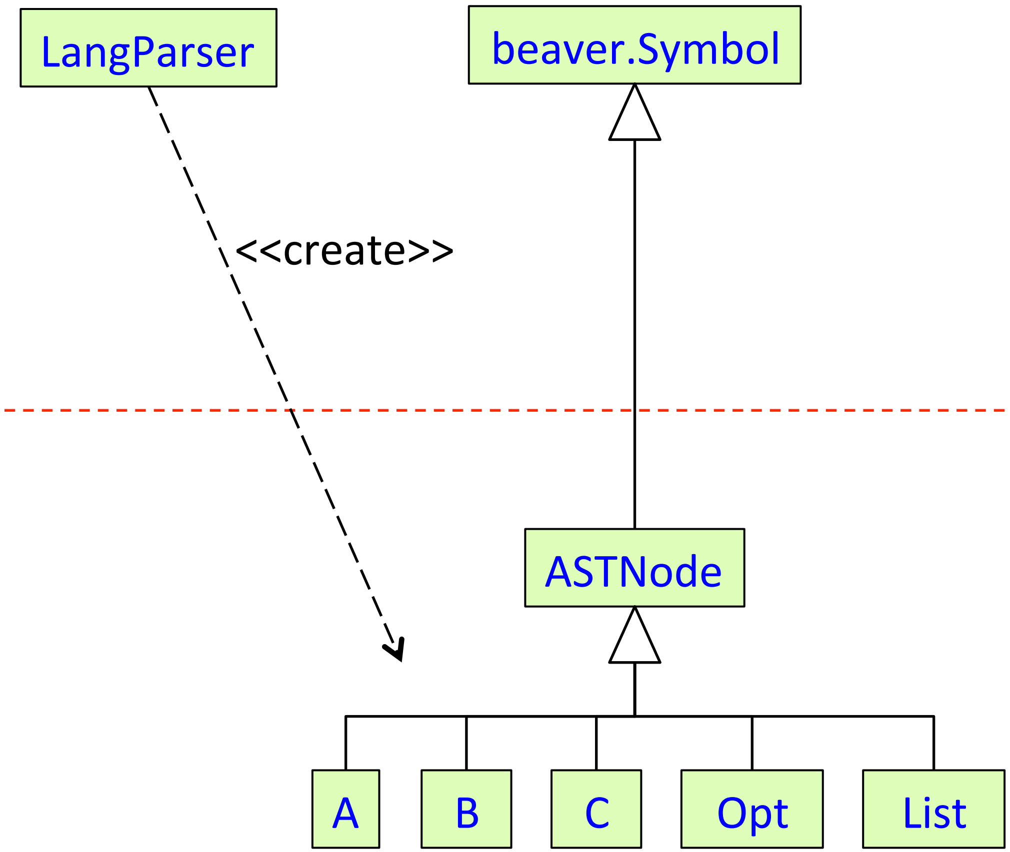
Beaver

Beaver spec

```
a = b [c];   {: return new A... :}
b = ... ;    {: return new B... :}
c = ... ;    {: return new C... :}
```

JastAdd abstract grammar

```
A ::= B [C];
B ::= ...;
C ::= ...;
```



JastAdd

Defining an abstract grammar

This is object-oriented modeling!

- What kinds of objects are there in the AST?
  E.g., Program, WhileStmt, Assignment, Add, ...

- What are the generalized concepts (abstract classes)?
  E.g., Statement, Expression, ...

- What are the components of an object?
  E.g., an Assignment has an Identifier and an Expression...

```
Program ::= ...;
abstract Statement;
abstract Expression;
WhileStmt : Statement ::= ...;
Assignment : Statement ::= Identifier Expression;
...
```

Use good names!

when you write...	**...the following should make sense**
A : B ::= ...	An A is a special kind of B
C ::= D E F	A C has a D, an E, and an F
D ::= X:E Y:E	A D has one E called X and another E called Y
G ::= [H]	A G may have an H
J ::= <K:T>	A J has a K token of type T
L ::= M*	An L has zero or more Ms

Examples of bad naming (from inexperienced programmers)	**Good naming**
A ::= [OptParam]; OptParam ::= Name Type;	A ::= [Param]; Param ::= Name Type;
A ::= Stmts*; abstract Stmts; While : Stmts ::= Exp Stmt;	A ::= Stmt*: abstract Stmt; While : Stmt ::= Exp Stmt;

Design simple abstract grammars!

- Abstract grammars should be clear and simple
- Don't let parsing details creep into the abstract grammar

Bad abstract grammar (parsing inspired)	**Good abstract grammar (simple, conceptual)**
A ::= First:B Rest:B*	A ::= B*
Add : Exp ::= Left:Exp Right:Term	Add : Exp ::= Left:Exp Right:Exp

- "At least one child" can easily be checked by a semantic check. Don't impose a more complex structure just to check this.
- Term, Factor, etc. is a parsing issue, not an abstract grammar issue.

Design a parsing grammar

- Design the abstract grammar first.

- Then design a high-level concrete grammar, making it as similar as possible to the abstract grammar.
    - Replace inheritance with alternative productions
    - The grammar will probably be ambiguous

- Then design a low-level concrete grammar, suitable for a particular parsing algorithm/tool.
  For Beaver:
    - Eliminate ambiguities
    - Eliminate repetition and optionals (will make it easier to construct the AST)

Semantic actions in parsers

- Code that is added to a parser, to perform actions during parsing.
- Usually, to build the AST.
- Old-style 1-pass compilers did the whole compilation as semantic actions.
- Parser generators support semantic actions in the parser specification.

Beaver example

Abstract grammar

```
abstract Stmt;
IfStmt : Stmt ::= Expr Stmt;
Assignment : Stmt ::= IdExpr Expr;
IdExpr : Expr ::= <ID:String>;
```

High-level CFG

```
stmt -> ifStmt | assignment
ifStmt -> IF "(" expr ")" stmt
assignment -> ID ASSIGN expr
```

beaver spec without semantic actions:

```
%class "LangParser";
%package "lang";
...
%terminals IF, LPAREN, RPAREN, ID, ASSIGN;

%goal stmt; // The start symbol

// Context-free grammar
stmt = ifStmt | assignment;
ifStmt = IF LPAREN expr RPAREN stmt;
assignment = ID ASSIGN expr;
```

Beaver example

Abstract grammar

```
abstract Stmt;
IfStmt : Stmt ::= Expr Stmt;
Assignment : Stmt ::= IdExpr Expr;
IdExpr : Expr ::= <ID:String>;
```

beaver spec with semantic actions:

```
%class "LangParser";
%package "lang";
...
%terminals IF, LPAREN, RPAREN, ID, ASSIGN;

%goal stmt; // The start symbol

%typeof stmt = "Stmt";
%typeof ifStmt = "IfStmt";
%typeof assignment = "Assignment";

// Context-free grammar
stmt = ifStmt | assignment;
ifStmt = IF LPAREN expr.e RPAREN stmt.s; {: return new IfStmt(e, s); :}
assignment =
   ID.id ASSIGN expr.e; {: return new Assignment(new IdExpr(id),e); :}
```

semantic actions build the trees

variables capture token strings and subtrees for nonterminals

the nonterminals return objects of the abstract grammar classes

Summary questions: Abstract syntax trees

- What is the difference between an abstract and a concrete syntax tree?
- What is the difference between an abstract and a concrete grammar?
- What is the correspondence between an abstract grammar and an object-oriented model?
- Orientation about JastAdd abstract grammars, traversal API, and connection to Beaver.
- What are properties of a good abstract grammar?
- What is a "semantic action"?
- How can Beaver be used for building ASTs?