Compiler Construction

More LL parsing
Abstract syntax trees

Lennart Andersson

Revision 2011–01–31

2010

LL(k)

Related names

top-down the parse tree is constructed top-down

recursive descent if it is implemented using one (recursive) method for each nonterminal

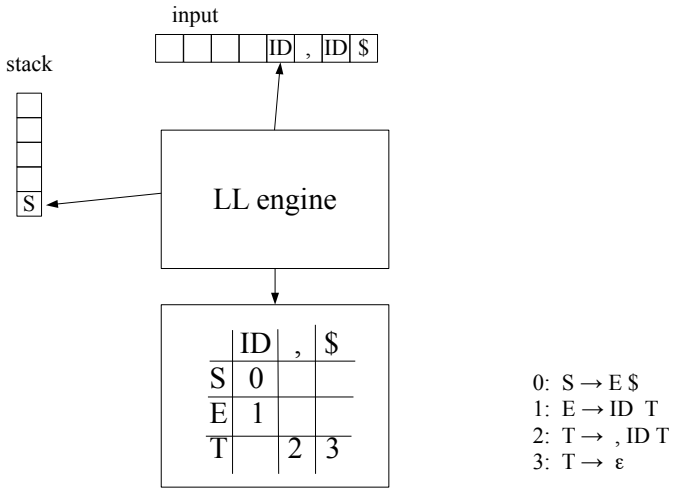
LL **L**eft-to-right scanning of input using **L**eftmost derivation

predictive the next k tokens are used to predict (decide) which production to use. k is often 1.

Top-down Parsing

- recursive descent
- easy to understand and to implement
- most ambiguities need to be treated by rewriting the grammar
- cannot handle some common situations (left-recursion, common prefixes). In these cases the grammar needs to be rewritten slightly.

LL(1) parsing engine



Left recursive grammar

p_1 : *term* → *term* `"*"` *factor*
p_2 : *term* → *factor*
p_3 : *factor* → *ID*
p_4 : *factor* → *INT*

	NLBL	*FIRST*
term	*F*	*ID INT*
factor	*F*	*ID INT*

	*	*ID*	*INT*
term		p_1 p_2	p_1 p_2
factor		p_3	p_4

A left recursive grammar is not LL(k).

Choice points in EBNF grammar

- multiple productions for some nonterminal
- alternatives |
- optionals [ ] or ?
- iteration $(\ldots)^*$ or $(\ldots)^+$

The recursive descent parser must be able to make the correct choice looking at the next token (or the next k tokens).

If JavaCC suggests using more lookahead then compute FIRST and FOLLOW sets at the choice point before following the advice.

Grammar with common prefix

p_1 : *expr* → *factor* `"*"` *expr*
p_2 : *expr* → *factor*
p_3 : *factor* → *ID*
p_4 : *factor* → *INT*
p_5 : *factor* → `"("` *expr* `")"`

	NLBL	*FIRST*
expr	*F*	*ID INT (*
term	*F*	*ID INT (*
factor	*F*	*ID INT (*

	*	ID	INT	(	)
expr		p_1, p_2	p_1, p_2	p_1, p_2	
factor		p_3	p_4	p_5	

A grammar with common prefixes is not LL(1).
It may be LL(k). This one is not. Why?

Another example

p_1 : *statement* → *ID* `"("` *idList* `")"`
p_2 : *statement* → *ID* `"="` *expr*

	ID	(	)	=
statement	p_1, p_2			
...				

LL(2) table

p_1 : *statement* → *ID* `"("` *idList* `")"`
p_2 : *statement* → *ID* `"="` *expr*

	ID ID	ID (	ID (	ID =	((	...
statement		p_1		p_2		
...						

LL(1) with local lookahead

p_1 : *statement* → *ID* `"("` *idList* `")"`
p_2 : *statement* → *ID* `"="` *expr*

	ID		(	)	=
statement	(	=			
	p_1	p_2			
...					

JavaCC

- can handle general LL(k)
- k=1 is the default *global lookahead*
- $k \geq 2$ must be specified; may be inefficient
- *local lookahead* may be specified for individual productions; may improve readability at a reasonable cost.
- local lookahead may be specified as number of tokens, grammatical expression, ...

LOOKAHEAD(k) in JavaCC

- JavaCC detects choice points where LOOKAHEAD(1) is not sufficient and suggests that the LOOKAHEAD should be increased
- After inserting LOOKAHEAD(k) with some k JavaCC will not check if this is sufficient but use the first rule that matches.
- Make sure that you understand what you are doing. If the grammar is ambiguous and you insert a LOOKAHEAD then JavaCC will not complain.

JavaCC example

```
void stmt() : {} {
   <ID> "(" idList() ")" ";"
|  <ID> <ASSIGN> expr()
}
```

```
> javacc PL0.jj
Reading from file PL0.jj . . .
Warning: Choice conflict involving two expansions at
         line 25, column 9 and line 26, column 9
         respectively. A common prefix is: <ID>
         Consider using a lookahead of 2 for earlier
         expansion.
Parser generated with 0 errors and 1 warnings.
>
```

How will the generated parser handle this conflict?

Use of local lookahead in JavaCC

```
void statement() : { } {
        LOOKAHEAD(2) <ID> "(" idList() ")" ";"
|       <ID> "=" expr()
|       ...
}
```

JavaCC will try the alternatives in order using lookahead 2 for the first one and lookahead=1 for the second ...

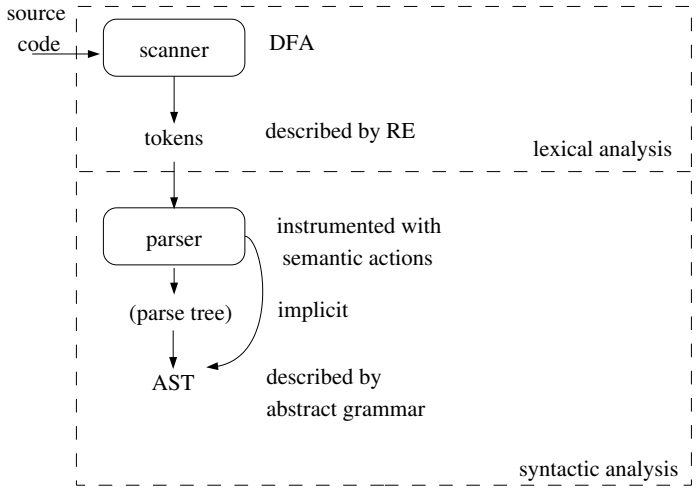
JavaCC dangling else

```
void ifStmt(): {} {
  <IF> expr <THEN> statement [<ELSE> statement]
}
```

```
Warning: Choice conflict in [...] construct at line 35,
         column 30. Expansion nested within construct
         and expansion following construct have common
         prefixes, one of which is: <ELSE> Consider using
         a lookahead of 2 or more for nested expansion.
```

How will the generated parser handle this conflict?
`LOOKAHEAD(1)` before `<ELSE>` removes the warning.

Syntactic analysis



What is an AST?

An abstract representation of a program

- revealing the structure
- suitable for analysis
- without unnecessary information required by for parsing
- independent of the parsing algorithm
- described by an abstract grammar

Abstract vs. concrete grammar

The concrete grammar describes the external representation, a string (sequence) of tokens.

- It defines a language.
- It is used backwards to find a derivation for a given string.
- It can be used to build an abstract representation.

The abstract grammar describes the internal representation, a labeled tree.

- It defines a data type.
- It is used forwards to build tree.

Concrete/Abstract grammar

$$
\begin{array}{lll}
\mathit{expr} & \rightarrow & \mathit{term}\ (\texttt{"+"}\ \ \mathit{term})^* \\
\mathit{term} & \rightarrow & \mathit{factor}\ (\texttt{"*"}\ \ \mathit{factor})^* \\
\mathit{factor} & \rightarrow & \mathit{ID} \mid \mathit{INT} \mid\ \ \texttt{"("}\ \ \mathit{expr}\ \ \texttt{")"}
\end{array}
$$

```
Add: Expr ::= Expr Expr
Mul: Expr ::= Expr Expr
IdExpr: Expr ::= ID
IntExpr: Expr ::= INT
```


Java model

```
abstract class Expr {
}
class Add extends Expr {
  Expr expr1, expr2;
}
class Mul extends Expr {
  Expr expr1, expr2;
}
class IdExpr extends Expr {
  String ID;
}
class IntExpr extends Expr {
  String INT;
}
```

Java model with computation

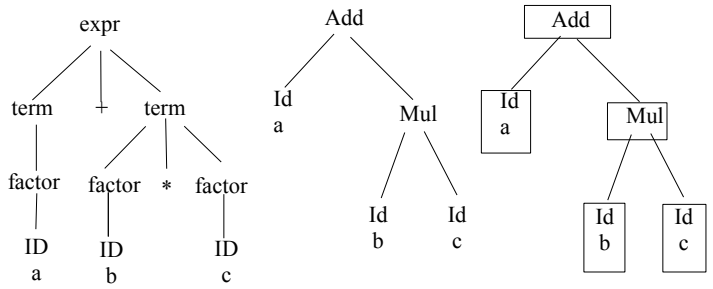
```
class Add extends Expr {
  Expr expr1, expr2;
  String toString(){
    return expr1.tostring() + expr2.toString();
  }
}
class IdExpr extends Expr {
  String ID;
  String toString(){
    return ID;
  }
}
```


JastAdd specification

```
abstract Expr;
Add: Expr     ::= Expr Expr;
Mul: Expr     ::= Expr Expr;
IdExpr: Expr  ::= <ID>;
IntExpr: Expr ::= <INT>;
```

It is easy to write a program that converts a jastadd specification to Java classes.

Parse tree, AST, Object diagram



parse tree AST object diagram

Example, grammar

CFG (EBNF):

program → "begin" [*stmt* (";" *stmt*)*] "end"
stmt → *ID* "=" *expr*
stmt → "if" *expr* "then" *stmt* ["else" *stmt*]
expr → *ID* | *INT*

JastAdd grammar

```
Program ::= Stmt*;
abstract Stmt;
Assinment: Stmt ::= <ID> Expr;
IfStmt: Stmt ::= Expr Then: Stmt [Else: Stmt];
abstract Expr;
IdExpr: Expr ::= <ID>;
IntExpr: Expr ::= <INT>;
```

Generated Java classes

```
Program ::= Stmt*;
abstract Stmt;
Assignment: Stmt ::= IdExpr Expr;
```

```
class Program extends ASTNode {
  int getNumStmt();
  Stmt getStmt(int i);
}
abstract class Stmt extends ASTNode;
class Assignment extends Stmt {
  IdExpr getIdExpr();
  Expr getExpr();
}
```


Generated Java classes

```
IfStmt: Stmt ::= Expr Then:Stmt [Else:Stmt];
```

```
class IfStmt extends Stmt {
  Expr getExpr();
  Stmt getThen();
  boolean hasElse();
  Stmt getElse();
}
```


Generated Java classes

```
abstract Expr;
IdExpr: Expr ::= <ID:String>;
IntExpr: Expr ::= <INT:String>;
```

```
abstract class Expr extends ASTNode;
class IdExpr extends Expr {
  String getID();
}
class IntExpr extends Expr {
  String getINT();
}
```


ASTNode and SimpleNode

```
class ASTNode extends SimpleNode {
  int getNumChild();
  ASTNode getChild(int i);
  ASTNode getParent();
}

class SimpleNode implements Node {
  void dump(String prefix);
}
```

Ambiguity

Can an abstract grammar be ambiguous?

- No, an abstract grammar is essentially a data type
- In Java it is implemented by a set of classes that can be used to build ASTs.
- Different trees are just different.

JastAdd notation — overview

notation	generated Java code
A1;	class A1 extends ASTNode {}
A2: B;	class A2 extends B {}

A3: B ::= C D;	class A3 extends B {
	C getC(){...}
	D getD(){...}
	}

abstract A4 ...;	abstract class A4 ...

A5: B ::= [C];	class A5 extends B {
	boolean hasC(){...}
	C getC(){...}
	}

A6: B ::= C*;	class A6 extends B {
	int getNumC(){...}
	C getC(int i){...}
	}

A7: B ::= <C: D>;	class A7 extends B {
	D getC(){...}
	void setC(D d){...}
	}

JastAdd notation — overview

notation	generated Java code

A8: B ::= Left: C Right: C;	class A8 extends B {
	C getLeft(){...}
	C getRight(){...}
	}

A10: A9 ::= ...;	class A10 extends A9 { }

A11: A ::= B C* [D];	class A11 extends B {
	B getB(){...}
	int getNumC(){...}
	...
	}

Other JastAdd classes

generated Java code
class ASTNode extends SimpleNode { int getNumChild(){...} ASTNode getChild(int index){...} ASTNode getParent(){...}
class List extends ASTNode {...}
class Opt extends ASTNode {...}

- Avoid using `getChild(int)`!
- `getChild` and `getParent` will traverse List and Opt nodes explicitly.
- With `Item: A ::= A B* [C]`
- `item.getB(2)` and `item.getC()` are much clearer than
- `(B) item.getChild(1).getChild(2)` and
- `(C) item.getChild(2).getChild(0)`

Inheriting right-hand side structure

Useful for, e.g., binary operations and the Template Method design pattern:

```
abstract BinExpr : Expr ::= Left:Expr Right:Expr;
Add : BinExpr;
Sub : BinExpr;
...
```


Defining an abstract grammar

- This is object-oriented modeling.
- Which are the objects? Program, statement, statement list, assignment, if-statement, expression, identifier, numeral, a sum (+ with two operands), ...
- Which are the generalizations (abstract classes)? Statement, expression, ...
- Which are the components of an object? An assignment has an identifier and an expression. An if-statement has an expression and two statement lists, an *then* part and an *else* part.

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
E ::= [F]	an E may have an F
F ::= <K:T>	an F has a token called K with a value of type T
L ::= M*	an L has a number of Ms

Recommendations

- Keep things together reflecting the logical structure.
- Keep the rules simple.
- Introduce new classes for structures that appear in several places.
- Don't try to model restrictions that are better implemented by the concrete grammar or later semantic checks. A list with at least one element should be modeled by a simple list.

Defining a concrete LL grammar

- Define the abstract grammar before the concrete grammar if this is an option.
- Use * and [] in favor of recursion and ϵ.
- For high level constructs like procedures and statements introduce keywords and separators to make parsing simple and unambiguous, LL(1). There will often be one grammar production for each AST class; the abstract classes will correspond to a selection of the alternatives for the subclasses.
- For low level constructs, primarily expressions, the concrete grammar should instead be based upon a verbal description making precedences clear: An *expression* is a sequence of one or more *term*s separated by + or - signs. A *term* is a sequence of one or more *factor*s separated by * or / signs, etc.

. . . Defining a concrete LL grammar

- Focus on recognizing the *structure* of the program to make the AST generation easy.
- Don't try to detect semantic errors such as type errors with the CFG. It is usually impossible to detect all of them with the grammar.
- A parsing grammar will often be much simpler than a language description to be read by humans, since the latter will often describe the grammar and the meaning of constructs at the same time.

Building the AST

- A grammar production can be augmented with *semantic actions*.
- A semantic action is a piece of code that is executed when the production is used.
- Semantic actions can be used to build the AST.
- A parser generator may have special support for building ASTs.

Hand-coded parser without actions

```
void stmt() {
  switch(token) {
  case IF:
    accept(IF); expr(); accept(THEN); stmt();
    break;
  case ID:
    accept(ID); accept(EQ); expr();
    break;
  default:
    error();
  }
}
```

Hand-coded parser with semantic actions

```
Stmt stmt() {
  switch(token.kind) {
  case IF:
    accept(IF);
    Expr e = expr();
    accept(THEN);
    Stmt s = stmt();
    return new IfStmt(e, s);
  case ID:
    IdExpr id = new IdExpr(token.image);
    accept(ID);
    accept(EQ);
    Expr e = expr();
    return new Assignment(id, e);
  }
}
```


.jj file with actions

```
Stmt stmt() : {Stmt s;}
{
  (s = ifStmt() | s = assignment()) {return s;}
}
IfStmt  ifStmt() : {Expr e; Stmt s;}
{
  "if" e = expr() "then" s = stmt()
  {return new IfStmt(e, s);}
}
Assignment assignment() : {Token t; Expr e;}
{
  t = <ID> "=" e = expr()
  {return new Assignment(new IdExpr(t.image), e);}
}
```


Building trees with JJTree

- jjtree is a preprocessor to javacc using a .jjt file
- grammar productions with tree building directives "#..."
- jjtree generates a .jj file with tree building actions
- builds a parse tree by default (Assignment 2)
- with option NODE_DEFAULT_VOID=true building requires # directives

jjtree stack

jjtree has a stack for collecting the children of a node. It is easy to

- build a node corresponding to a nonterminal
- ignore building a node for a terminal/nonterminal
- build *List* and *Opt* nodes
- build "left recursive" trees when the grammar is right recursive
- build trees when the grammar has common prefixes

Building an AST node for a nonterminal

Add a # directive to the corresponding AST class

```
void ifStmt() #IfStmt : { } {
  <IF> expr() <THEN> stmt() <FI>
}

IfStmt: Stmt ::= Expr Stmt
```

- the current top of stack is marked
- *expr()* pushes an Expr node
- *stmt()* pushes a Stmt node
- a new IfStmt node is created
- the nodes above the mark will be popped and
- become children of the IfStmt node
- this node will be pushed on the stack

Skipping nodes

A terminal or nonterminal without a # directive will be skipped

```
void ifStmt(): { } {
  <IF> expr() <THEN> stmt() <FI>
}
```

- *expr()* pushes an Expr node if it has a # directive
- no node is pushed for `THEN`
- *stmt()* pushes a Stmt node if it has a # directive
- no nodes will be popped by this production
- no IfStmt is created
- this is probably an error

Building a token node

```
IdExpr: Expr ::= <ID:String>;


void idExpr() #IdExpr : {Token t;}
{
  t = <ID>
  {jjtThis.setID(t.image);}
}
```

- `jjtThis` refers to the new IdExpr node
- the `setID(String)` method has been generated by jastadd

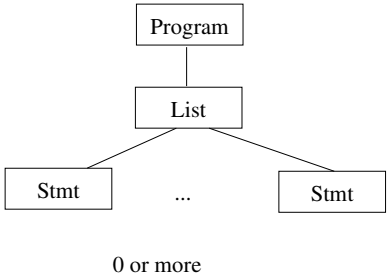
Building a List node

```
void program() #Program : {}
{
  "begin" ([stmt() (";" stmt())*]) #List(true) "end"
}
Program ::= Stmt*;
```

- a marker for Program is put on the stack
- `"begin"` is parsed without pushing anything
- a marker for List is put on the stack
- a node is pushed on the stack for each stmt
- the parser reaches #List(true) and all nodes above the List marker are popped and become children of a new List node
- this node is pushed on the stack
- the `"end"` token is accepted
- the only node above the Program mark is popped an becomes the child of a new Program node
- this node is pushed on the stack

Why #List(true)?

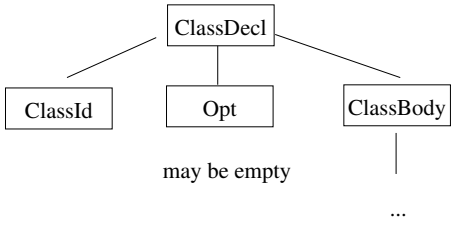
- the List will be built even it is empty
- with `#List(false)` nothing will be build for an empty list
- with jastadd empty trees must be generated



Optional node

```
void classDecl() #ClassDecl : { }
  "class" classId() (["extends" classId()]) #Opt(true)
   ClassBody()
```

- an Opt node will be built even if there is no extends clause
- with jastadd empty Opt nodes must be generated



Building left-recursive trees with jjtree

```
abstract Expr;
BinExpr: Expr ::= Left:Expr Right:Expr;
Mul: BinExpr;
Div: BinExpr;

void expr() : { }
{
  factor()
  ( "*" factor() #Mul(2)
  | "/" factor() #Div(2))*
}
```

- `#Mul(2)` builds a Mul node
- pops two nodes from the stack
- and turns them into children
- pushes the new Mul node to the stack

Returning the AST to the client

Creating and using the parser

```
Parser parser = new Parser(...);
Start start = parser.start();
```

jjtree specification

```
Start start() #Start : { }
{
  expr();
  {return jjtThis;}
}
```

`jjtThis` refers to the `Start` node created by `start()`