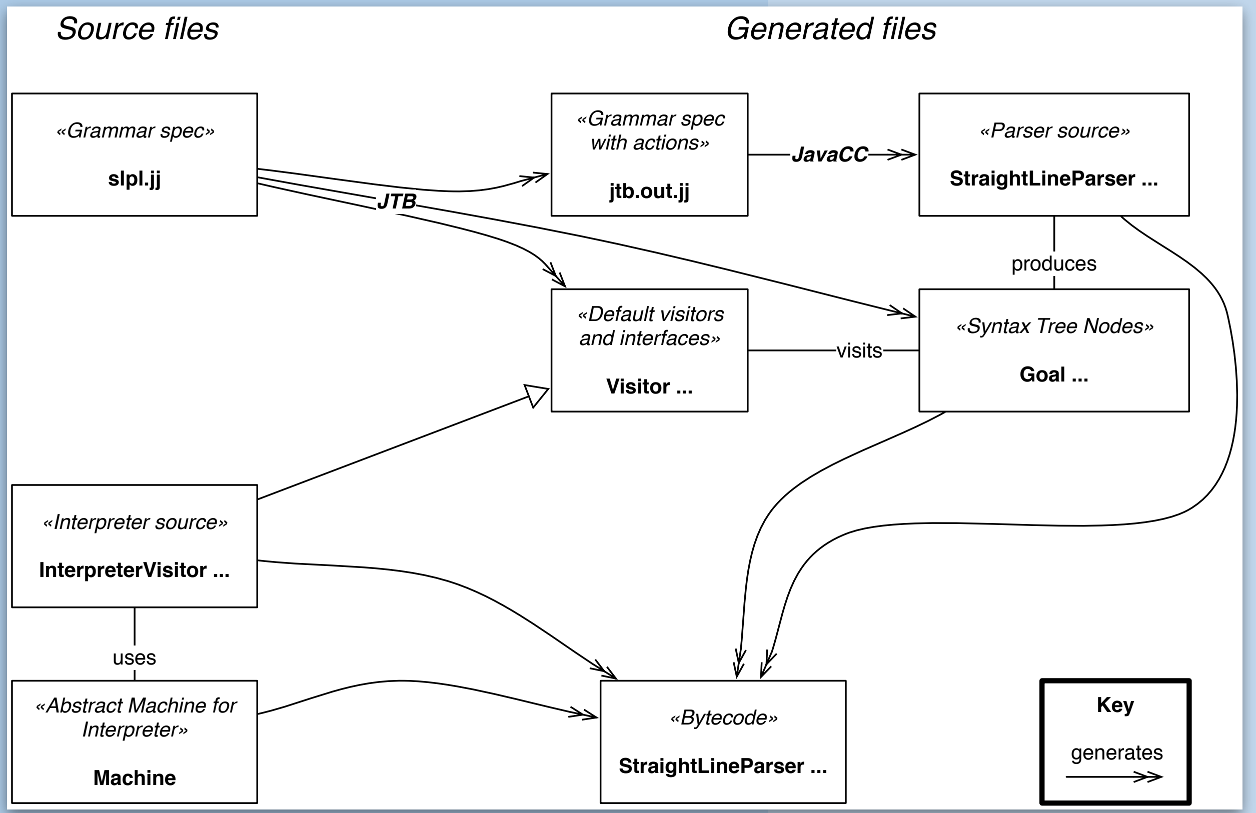
4. Parsing in Practice

Oscar Nierstrasz



Thanks to Jens Palsberg and Tony Hosking for their kind permission to reuse and adapt the CS132 and CS502 lecture notes.
http://www.cs.ucla.edu/~palsberg/
http://www.cs.purdue.edu/homes/hosking/

Roadmap

> Bottom-up parsing
> LR(k) grammars
> JavaCC, Java Tree Builder and the Visitor pattern
> Example: a straightline interpreter

See, *Modern compiler implementation in Java* (Second edition), chapters 3-4.

Roadmap

> **Bottom-up parsing**
> LR(k) grammars
> JavaCC, Java Tree Builder and the Visitor pattern
> Example: a straightline interpreter

Some definitions

Recall:

> For a grammar G, with start symbol S, any string α such that $S \Rightarrow^* \alpha$ is called a *sentential form*
>
> — If $\alpha \in V_t^*$, then α is called a *sentence* in L(G)
>
> — Otherwise it is just a sentential form (not a sentence in L(G))

> A *left-sentential form* is a sentential form that occurs in the leftmost derivation of some sentence.

> A *right-sentential form* is a sentential form that occurs in the rightmost derivation of some sentence.

Example — rightmost derivation

The left-recursive expression grammar (*original form)*

1.	<goal>	::=	<expr>
2.	<expr>	::=	<expr> + <term>
3.		|	<expr> – <term>
4.		|	<term>
5.	<term>	::=	<term> * <factor>
6.		|	<term> / <factor>
7.		|	<factor>
8.	<factor>	::=	`num`
9.		|	`id`

Prod'n.	Sentential Form
–	$\langle\text{goal}\rangle$
1	$\underline{\langle\text{expr}\rangle}$
3	$\underline{\langle\text{expr}\rangle - \langle\text{term}\rangle}$
5	$\langle\text{expr}\rangle - \underline{\langle\text{term}\rangle * \langle\text{factor}\rangle}$
9	$\langle\text{expr}\rangle - \langle\text{term}\rangle * \underline{\texttt{id}}$
7	$\langle\text{expr}\rangle - \underline{\langle\text{factor}\rangle} * \texttt{id}$
8	$\langle\text{expr}\rangle - \underline{\texttt{num}} * \texttt{id}$
4	$\underline{\langle\text{term}\rangle} - \texttt{num} * \texttt{id}$
7	$\underline{\langle\text{factor}\rangle} - \texttt{num} * \texttt{id}$
9	$\underline{\texttt{id}} - \texttt{num} * \texttt{id}$

How do we parse (bottom-up) to arrive at this derivation?

```
x – 2 * y
```

Note that here we see the derivation, not the parse.

Now we will see how bottom-up parsing allows us to recover this derivation.

Bottom-up parsing

Goal:

—Given an input string w and a grammar G, construct a parse tree by starting at the leaves and working to the root.

> The parser repeatedly matches a *right-sentential form* from the language against the tree's upper frontier.

> At each match, it applies a *reduction* to build on the frontier:

—each reduction matches an upper frontier of the partially built tree to the RHS of some production

—each reduction adds a node on top of the frontier

> The final result is a *rightmost derivation*, in reverse.

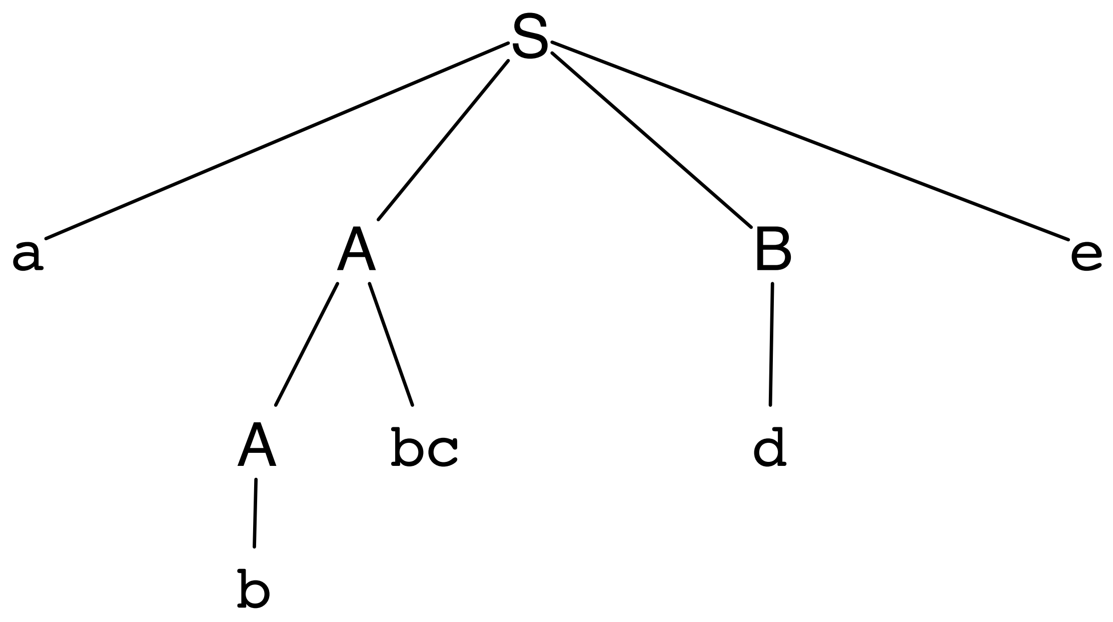
Example

Consider the grammar:

1.	S	→	aABe
2.	A	→	Abc
3.		|	b
4.	B	→	d

and the input string: abbcde

Sentential Form	Action
abbcde	shift a
abbcde	no match; shift b
a**b**bcde	match; reduce (3)
a**A**bcde	no match; shift b
aA**b**cde	lookahead ⇒ shift c
a**Abc**de	match; reduce (2)
aA**d**e	shift d
aA**B**e	match; reduce (4)
aABe	shift e
S	match; reduce (1)



A “shift” step advances the input stream by one token and creates a one-token parse-tree.

A “reduce” step applies a grammar rule to one or more trees on the “stack”, joining them and replacing them by a new parse tree for the non-terminal on the left-hand side of the rule.

We parse bottom-up, replacing terms by non-terminals.

Reading in reverse, we have a rightmost derivation, first replacing S, then B, A and A again.

Note that you have more context than with top-down since you may have a whole AST on the stack (A).

Handles

> A ***handle*** of a right-sentential form γ is a production $A \rightarrow \beta$ and a position in γ where β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ

—Suppose: $S \Rightarrow^* \alpha A w \Rightarrow \alpha\beta w$

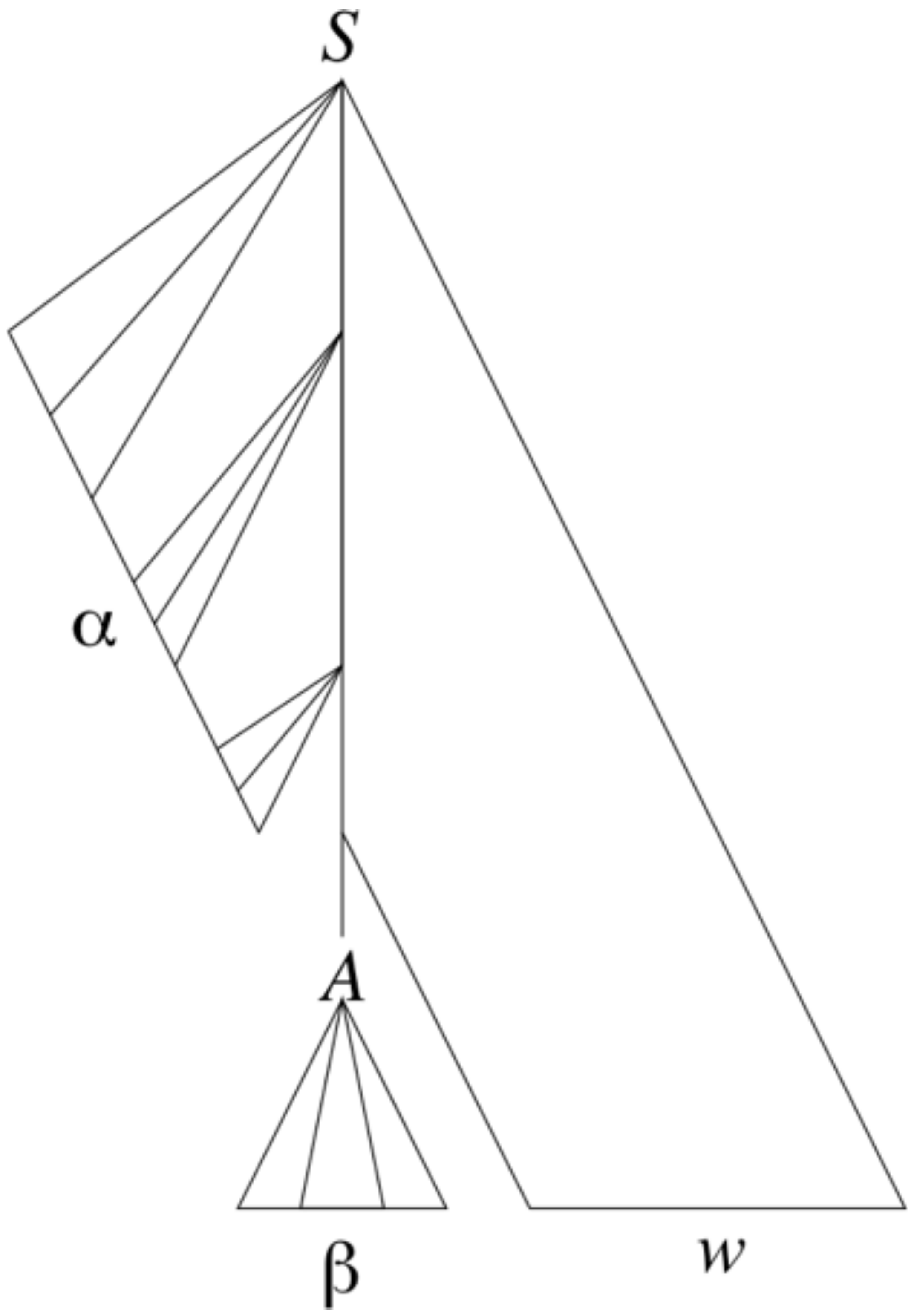
—Then $A \rightarrow \beta$ in the position following α is a *handle* of $\alpha\beta w$

NB: Because γ is a right-sentential form, the substring to the right of a handle contains *only terminal symbols*.

The handles in our previous example correspond to the points where we prune (reduce): in `aAbcde`, the handle is rule A → A`bc` in the position following “`a`”.

Non-terminals are only to the left (the stack) since you are parsing left-to-right.

Handles



The handle A → β in the parse tree for αβw

Handles

> **Theorem:**

—If G is unambiguous then every right-sentential form has a unique handle.

> **Proof:** (by definition)

1. G is unambiguous $\Rightarrow$ rightmost derivation is unique
2. $\Rightarrow$ a unique production $A \rightarrow \beta$ applied to take γ_{i-1} to γ_i
3. $\Rightarrow$ a unique position k at which $A \rightarrow \beta$ is applied
4. $\Rightarrow$ a unique handle $A \rightarrow \beta$

Handle-pruning

The process to construct a bottom-up parse is called *handle-pruning*

To construct a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

we set i to n and apply the following simple algorithm:

```
For i = n down to 1
1. Find the handle $A_i \rightarrow \beta_i$ in $\gamma_i$
2. Replace $\beta_i$ with $A_i$ to generate $\gamma_{i-1}$
```

This takes 2n steps, where n is the length of the derivation

Stack implementation

- One scheme to implement a handle-pruning, bottom-up parser is called a *shift-reduce parser*.
- Shift-reduce parsers use a *stack* and an *input buffer*
  1. initialize stack with $
  2. Repeat until the top of the stack is the goal symbol and the input token is $
     a) *Find the handle.*
        If we don't have a handle on top of the stack, shift (push) an input symbol onto the stack
     b) *Prune the handle.*
        If we have a handle $A \rightarrow \beta$ on the stack, reduce
        I. Pop |β| symbols off the stack
        II. Push A onto the stack

NB: In practice we also lookahead to determine whether to shift or reduce!

Actually, this is an LR(0) parser algorithm, since no lookahead is used.

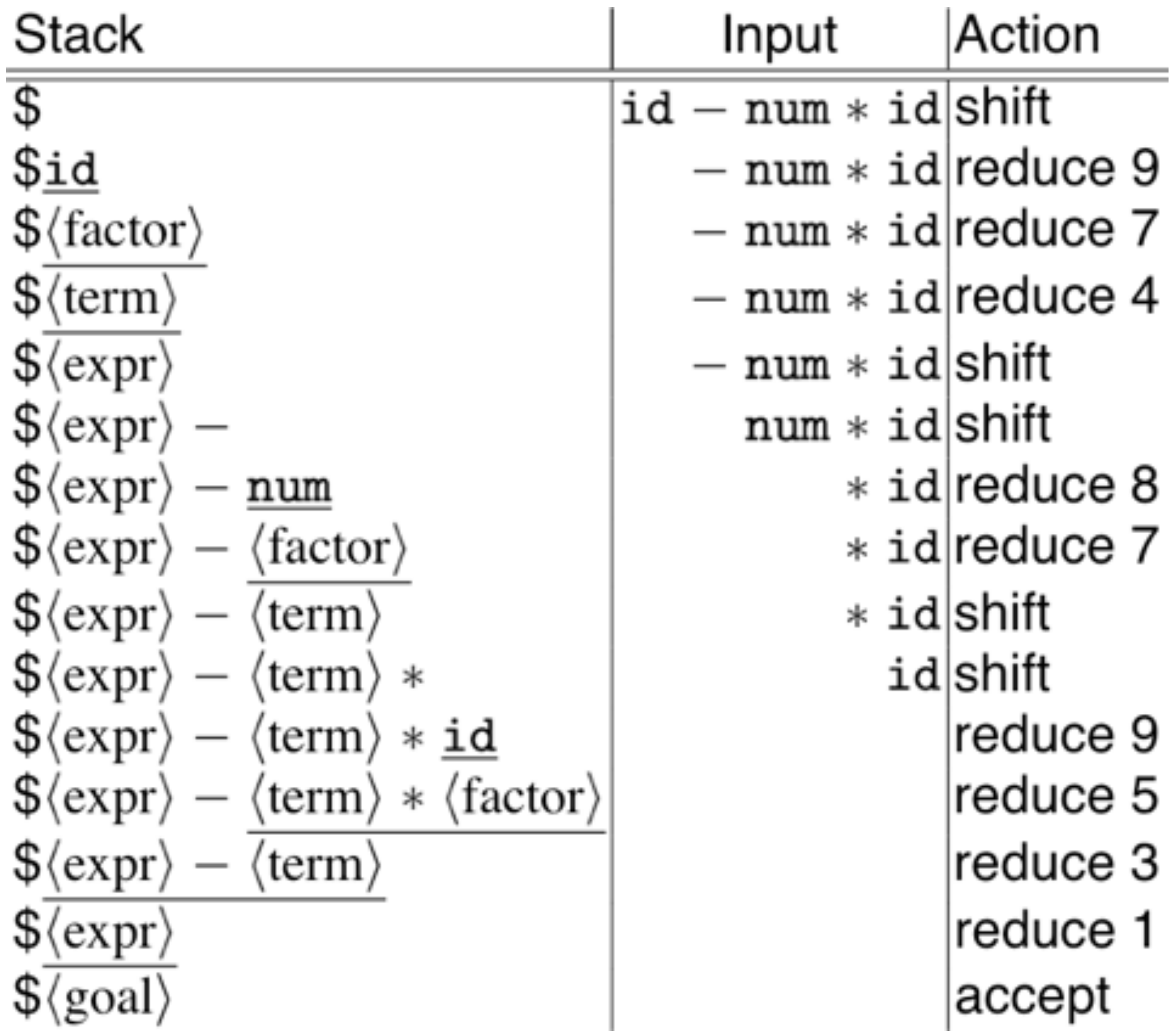
Shift-reduce parsing

A shift-reduce parser has just four canonical actions:

shift	next input symbol is shifted (pushed) onto the top of the stack
reduce	right end of handle is on top of stack; locate left end of handle within the stack; pop handle off stack and push appropriate non-terminal LHS
accept	terminate parsing and signal success
error	call an error recovery routine

Example: back to `x–2*y`

1.	<goal>	::=	<expr>
2.	<expr>	::=	<expr> + <term>
3.		|	<expr> – <term>
4.		|	<term>
5.	<term>	::=	<term> * <factor>
6.		|	<term> / <factor>
7.		|	<factor>
8.	<factor>	::=	`num`
9.		|	`id`



Stack	Input	Action
$\$$	$\texttt{id} - \texttt{num} * \texttt{id}$	shift
$\$\underline{\texttt{id}}$	$- \texttt{num} * \texttt{id}$	reduce 9
$\$\underline{\langle\text{factor}\rangle}$	$- \texttt{num} * \texttt{id}$	reduce 7
$\$\underline{\langle\text{term}\rangle}$	$- \texttt{num} * \texttt{id}$	reduce 4
$\$\langle\text{expr}\rangle$	$- \texttt{num} * \texttt{id}$	shift
$\$\langle\text{expr}\rangle -$	$\texttt{num} * \texttt{id}$	shift
$\$\langle\text{expr}\rangle - \underline{\texttt{num}}$	$* \texttt{id}$	reduce 8
$\$\langle\text{expr}\rangle - \underline{\langle\text{factor}\rangle}$	$* \texttt{id}$	reduce 7
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle$	$* \texttt{id}$	shift
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle *$	$\texttt{id}$	shift
$\$\langle\text{expr}\rangle - \langle\text{term}\rangle * \underline{\texttt{id}}$		reduce 9
$\$\langle\text{expr}\rangle - \underline{\langle\text{term}\rangle * \langle\text{factor}\rangle}$		reduce 5
$\$\underline{\langle\text{expr}\rangle - \langle\text{term}\rangle}$		reduce 3
$\$\underline{\langle\text{expr}\rangle}$		reduce 1
$\$\langle\text{goal}\rangle$		accept

1. *Shift until top of stack is the right end of a handle*
2. *Find the left end of the handle and reduce*

5 shifts + 9 reduces + 1 accept

After reducing <factor> to <term> we can either reduce again to <goal> or shift. A single token lookahead tells us to shift (since otherwise we would have reached our goal without consuming all the input).

Similarly after reducing <factor> to <term> we can either reduce <expr>`-`<term> or shift `*`. Again, the lookahead token tells us to shift rather than reduce (since otherwise we would be left with <expr>`*` on the stack, which can never be reduced!).

Here we only sketch out the general idea of bottom-up parsing.

What is not covered:

- How to recognise handles?
- How to use (and compute) lookahead tokens?

Roadmap

> Bottom-up parsing
> **LR(k) grammars**
> JavaCC, Java Tree Builder and the Visitor pattern
> Example: a straightline interpreter

LR(k) grammars

A grammar G is LR(k) iff:

1. $S \Rightarrow_{rm}^{*} \alpha A w \Rightarrow_{rm} \alpha\beta w$
2. $S \Rightarrow_{rm}^{*} \gamma B x \Rightarrow_{rm} \alpha\beta y$
3. $\mathrm{FIRST}_k(w) = \mathrm{FIRST}_k(y) \Rightarrow \alpha A y = \gamma B x$

I.e., if $\alpha\beta w$ *and* $\alpha\beta y$ *have the same k-symbol lookahead, then there is a unique handle to reduce in the rightmost derivation.*

A grammar is LR(k) if k tokens of lookahead are enough to determine a unique parse:

Assume sentential forms $\alpha\beta w$ and $\alpha\beta y$, with common prefix $\alpha\beta$ and common k-symbol lookahead $FIRST_k(w) = FIRST_k(y)$, such that $\alpha\beta w$ reduces to $\alpha A w$ and $\alpha\beta y$ reduces to $\gamma B x$.

The common prefix means $\alpha\beta y$ also reduces to $\alpha A y$, for the same result, i.e., $\alpha A y = \gamma B x$, or $\alpha = \gamma$, $A = B$ and $y = x$.

Why study LR grammars?

LR(1) grammars are used to construct LR(1) parsers.

- everyone’s favorite parser
- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars parsable by a deterministic, bottom-up parser
- efficient parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by predictive (i.e., LL) parsers

LL(k): recognize use of a production $A \rightarrow \beta$ seeing first k symbols of β

LR(k): recognize occurrence of β (the handle) having seen all of what is derived from β plus k symbols of look-ahead

Left versus right recursion

> ***Right Recursion:***
>
> —needed for termination in predictive parsers
>
> —requires more stack space
>
> —right associative operators

> ***Left Recursion:***
>
> —works fine in bottom-up parsers
>
> —limits required stack space
>
> —left associative operators

> ***Rule of thumb:***
>
> —right recursion for *top-down parsers*
>
> —left recursion for *bottom-up parsers*

Parsing review

> **Recursive descent**
  - A hand coded recursive descent parser directly encodes a grammar (typically an LL(1) grammar) into a series of mutually recursive procedures. It has most of the linguistic limitations of LL(1).

> **LL(k):**
  - must be able to recognize the *use of a production* after seeing only the first k symbols of its right hand side.

> **LR(k):**
  - must be able to *recognize the occurrence of the right hand side of a production* after having seen all that is derived from that right hand side with k symbols of look-ahead.

> *The dilemmas:*
  - LL dilemma: pick $A \rightarrow b$ or $A \rightarrow c$? (which rule to apply?)
  - LR dilemma: pick $A \rightarrow b$ or $B \rightarrow b$? (how to reduce?)

Roadmap

> Bottom-up parsing
> LR(k) grammars
> **JavaCC, Java Tree Builder and the Visitor pattern**
> Example: a straightline interpreter

The Java Compiler Compiler

- “Lex and Yacc for Java.”
- Based on **LL(k)** rather than LR(1) or LALR(1).
- Grammars are written in EBNF.
- Transforms an EBNF grammar into an LL(k) parser.
- Supports embedded action code written in Java (just like Yacc supports embedded C action code)
- The look-ahead can be changed by writing `LOOKAHEAD(…)`
- The whole input is given in just one file (not two).

Aside: LALR parsers start with an LR(0) state machine and then compute lookahead *sets* for all rules in the grammar, checking for ambiguity.

The JavaCC input format

> Single file:
  - header
  - token specifications for lexical analysis
  - grammar

Examples

Token specification:

```
TOKEN : /* LITERALS */
{
  < INTEGER_LITERAL: ( ["1"-"9"] ([“0"-"9"])* | "0" ) >
}
```

Production:

Declarations

Productions and actions

```
void StmList() :
{}
{
  Stm() ( ";" Stm() ) *
}
```

NB: with Java Tree Builder, the actual declarations and actions are *inferred and generated* to build a default parse tree (i.e., a concrete syntax tree).

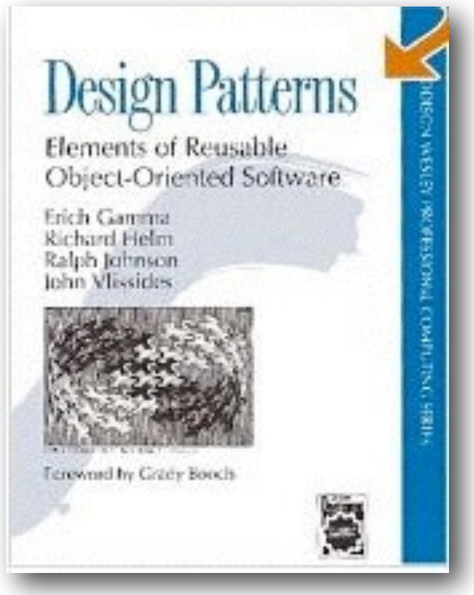
Generating a parser with JavaCC

```
javacc fortran.jj          // generates a parser
javac Main.java            // Main.java calls the parser
java Main < prog.f         // parses the program prog.f
```

NB: JavaCC is just one of many tools available …
See: http://catalog.compilertools.net/java.html

The Visitor Pattern

> *Intent:*
>
> —Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.



Sneak Preview

> When using the Visitor pattern,
—the set of classes must be fixed in advance, and
—each class must have an `accept` method.

First Approach: `instanceof` and downcasts

The running Java example: summing an integer list.

```
public interface List {}
public class Nil implements List {}
public class Cons implements List {
 int head;
 List tail;
 Cons(int head, List tail) {
  this.head = head;
  this.tail = tail;
 }
}
```

```
public class SumList {
 public static void main(String[] args) {
  List l = new Cons(5, new Cons(4,
    new Cons(3, new Nil())));
  int sum = 0;
  boolean proceed = true;
  while (proceed) {
   if (l instanceof Nil) {
    proceed = false;
   } else if (l instanceof Cons) {
    sum = sum + ((Cons) l).head;
    l = ((Cons) l).tail;
   }
  }
  System.out.println("Sum = " + sum);
 }
}
```

Advantage: The code does not touch the classes Nil and Cons.
Drawback: The code must use downcasts and `instanceof` to check what kind of List object it has.

This is a notorious code smell. You should be extremely suspicious of any code that needs to perform run-time type checks to downcast objects. It is usually a sign of poor object-oriented design, or a naive translation of procedural code to OO constructs.

Second Approach: Dedicated Methods

```
public interface List {
 public int sum();
}
public class Nil implements List {
 public int sum() {
  return 0;
 }
}
public class Cons implements List {
 int head;
 List tail;
 Cons(int head, List tail) {
  this.head = head;
  this.tail = tail;
 }
 public int sum() {
  return head + tail.sum();
 }
}
```

The classical OO approach is to offer dedicated methods through a common interface.

```
public class SumList {
 public static void main(String[] args) {
  List l = new Cons(5, new Cons(4,
         new Cons(3, new Nil())));
  System.out.println("Sum = "
   + l.sum());
 }
}
```

Advantage: Downcasts and `instanceof` calls are gone, and the code can be written in a systematic way.
Disadvantage: For each new operation on `List`-objects, new dedicated methods have to be written, and all classes must be recompiled.

Third Approach: The Visitor Pattern

> The Idea:

- —Divide the code into an object structure and a Visitor
- —Insert an `accept` method in each class. Each accept method takes a Visitor as argument.
- —A Visitor contains a `visit` method for each class (overloading!). A method for a class C takes an argument of type C.

NB: In a dynamically typed language you would introduce a `visitC` method for each class `C`, since it is not possible to have multiple methods in such languages with the same name, but taking different argument types.

Third Approach: The Visitor Pattern

```
public interface List {
 public void accept(Visitor v);
}
public class Nil implements List {
 public void accept(Visitor v) {
  v.visit(this);
 }
}
public class Cons implements List {
 int head;
 List tail;
 Cons(int head, List tail) {… }
 public void accept(Visitor v) {
  v.visit(this);
 }
}
public interface Visitor {
 void visit(Nil l);
 void visit(Cons l);
}
```

```
public class SumVisitor implements Visitor
{
 int sum = 0;
 public void visit(Nil l) { }

 public void visit(Cons l) {
  sum = sum + l.head;
  l.tail.accept(this);
 }

 public static void main(String[] args) {
  List l = new Cons(5, new Cons(4,
        new Cons(3, new Nil())));
  SumVisitor sv = new SumVisitor();
  l.accept(sv);
  System.out.println("Sum = " + sv.sum);
 }
}
```

NB: The visit methods capture both (1) actions, and (2) access of subobjects.

Note how in Java the type system is used to disambiguate the different `visit()` methods. In a dynamic language, there would instead be `visitNil()` and `visitCons()` methods.

Comparison

The Visitor pattern combines the advantages of the two other approaches.

	Frequent downcasts?	**Frequent recompilation?**
`instanceof` + downcasting	Yes	No
dedicated methods	No	Yes
Visitor pattern	No	No

JJTree (Sun) and Java Tree Builder (Purdue/UCLA) are front-ends for JavaCC that are based on Visitors

Visitors: Summary

> A visitor gathers related operations.
  - It also separates unrelated ones.

> Visitor makes adding new operations easy.
  - Simply write a new visitor.

> Adding new classes to the object structure is hard.
  - Key consideration: are you most likely to change the algorithm applied over an object structure, or are you most like to change the classes of objects that make up the structure?

> Visitor can break encapsulation.
  - Visitor’s approach assumes that the interface of the data structure classes is powerful enough to let visitors do their job. As a result, the pattern often forces you to provide public operations that access internal state, which may compromise its encapsulation.

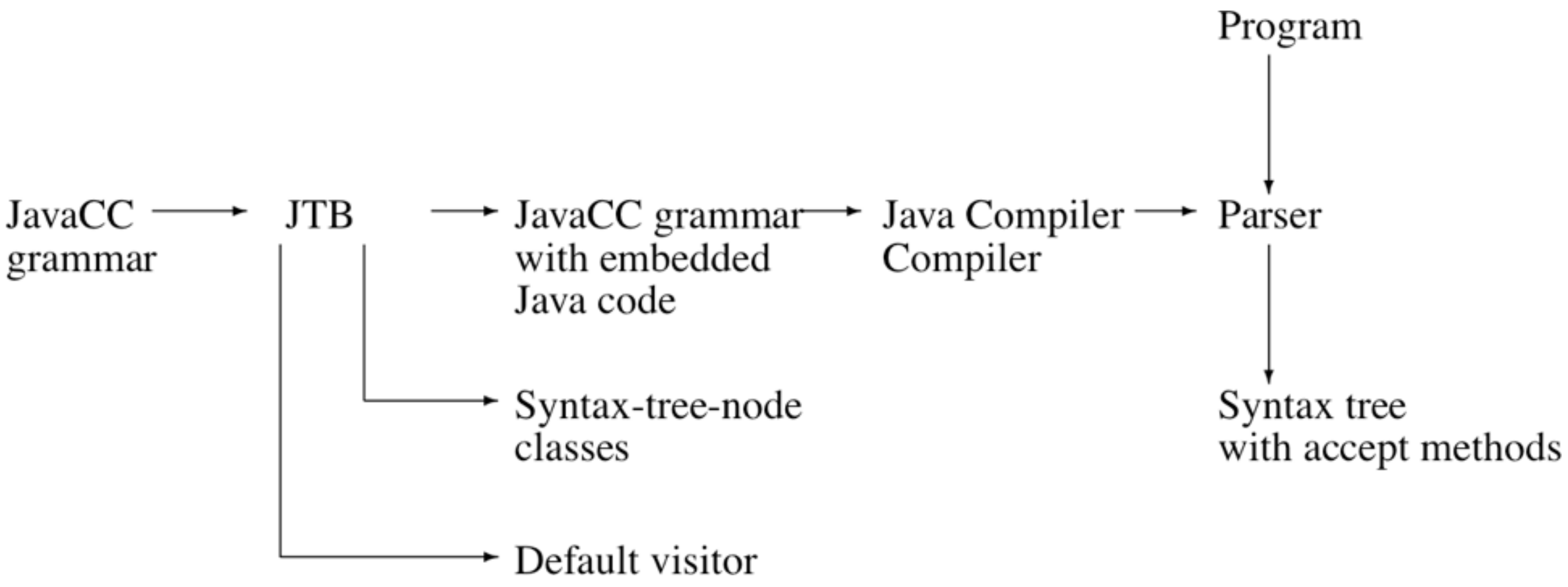
The Java Tree Builder (JTB)

- front-end for The Java Compiler Compiler.
- supports the building of syntax trees that can be traversed using visitors.
- transforms a bare JavaCC grammar into three components:
  - a JavaCC grammar with embedded Java code for building a syntax tree;
  - one class for every form of syntax tree node; and
  - a default visitor which can do a depth-first traversal of a syntax tree.

`http://compilers.cs.ucla.edu/jtb/`

The Java Tree Builder

The produced JavaCC grammar can then be processed by the Java Compiler Compiler to give a parser which produces syntax trees.
The produced syntax trees can now be traversed by a Java program by writing subclasses of the default visitor.



Using JTB

```
jtb fortran.jj         // generates jtb.out.jj
javacc jtb.out.jj      // generates a parser
javac Main.java        // Main.java calls the parser and visitors
java Main < prog.f     // builds a syntax tree and executes visitors
```

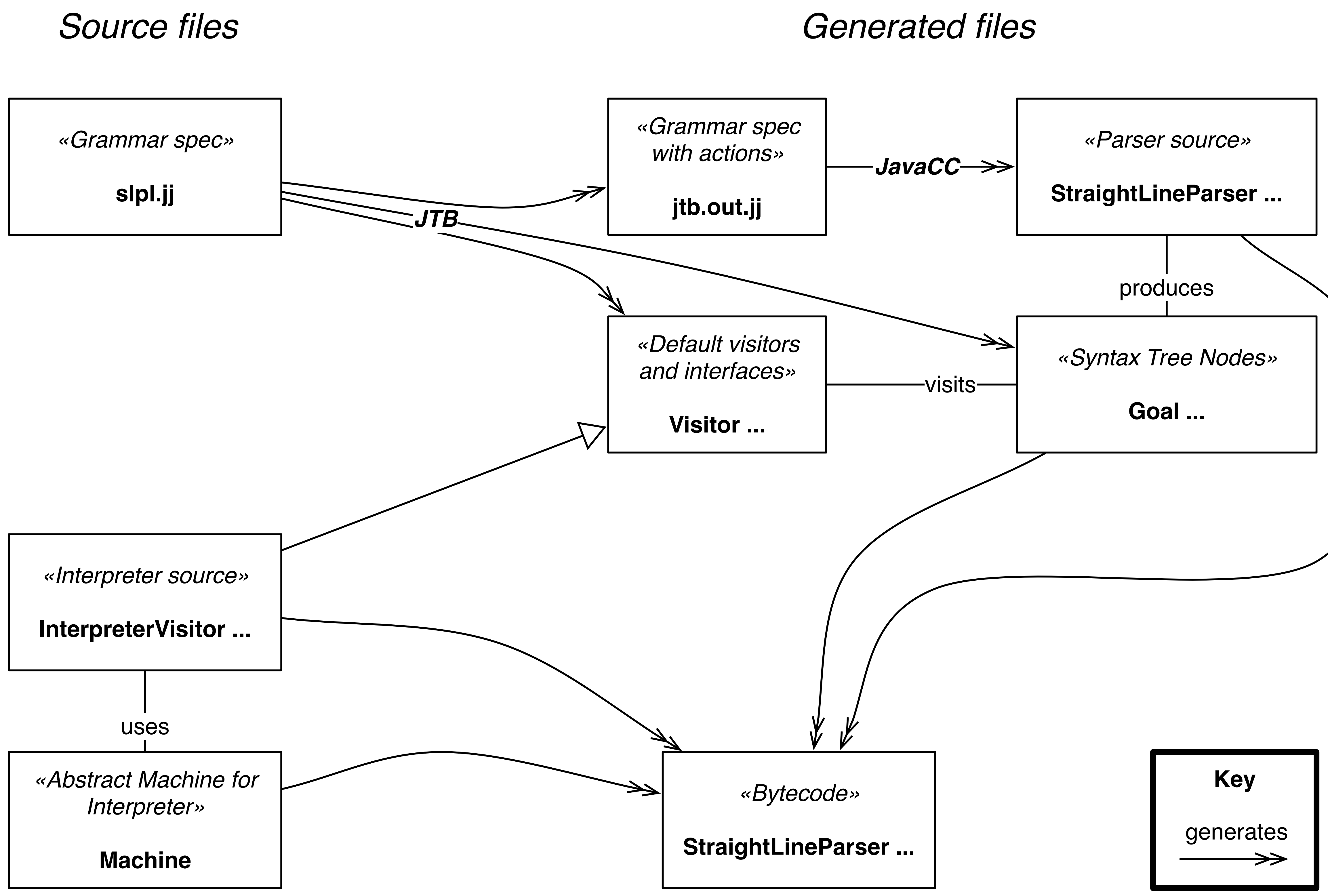
Roadmap

- Bottom-up parsing
- LR(k) grammars
- JavaCC, Java Tree Builder and the Visitor pattern
- **Example: a straightline interpreter**

Recall our straight-line grammar

Stm	→	Stm ; Stm	*CompoundStm*
Stm	→	`id` := Exp	*AssignStm*
Stm	→	`print` (ExpList)	*PrintStm*
Exp	→	`id`	*IdExp*
Exp	→	`num`	*NumExp*
Exp	→	Exp Binop Exp	*OpExp*
Exp	→	(Stm , Exp)	*EseqExp*
ExpList	→	Exp , ExpList	*PairExpList*
ExpList	→	Exp	*LastExpList*
Binop	→	+	*Plus*
Binop	→	–	*Minus*
Binop	→	×	*Times*
Binop	→	/	*Div*

Straightline Interpreter Files



JTB takes our unadorned grammar specification and expands it to a proper JavaCC grammar including generated actions to build a parse tree. Class for the syntax tree nodes are automatically generated, as are default visitors for those nodes.

Our interpreter will be a visitor that inherits from the generated one, and uses a simple "abstract machine" to interpret the parse tree nodes.

Recall that the source code is available here:

```
git clone git://scg.unibe.ch/lectures-cc-examples
```

Tokens

slpl.jj starts with the scanner declarations

```
options {
  JAVA_UNICODE_ESCAPE = true;
}

PARSER_BEGIN(StraightLineParser)
  package parser;
  public class StraightLineParser {}
PARSER_END(StraightLineParser)

SKIP : /* WHITE SPACE */
{ " " | "\t" | "\n" | "\r" | "\f" }

TOKEN :
{ < SEMICOLON: ";" >
| < ASSIGN: ":=" >
...
}
```

more tokens here!

```
TOKEN : /* LITERALS */
{ < INTEGER_LITERAL: ( ["1"-"9"] (["0"-"9"])*
| "0" ) >
}

TOKEN : /* IDENTIFIERS */
{ < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
| < #LETTER: [ "a"-"z", "A"-"Z" ] >
| < #DIGIT: ["0"-"9" ] >
}
```

Rewriting our grammar

Goal	→	StmList
StmList	→	**Stm (; Stm) ***
Stm	→	`id` := Exp
	|	`print` "(" ExpList ")"
Exp	→	**MulExp ((+ | –) MulExp) ***
MulExp	→	**PrimExp ((* | /) PrimExp) ***
PrimExp	→	`id`
	|	`num`
	|	"(" StmList , Exp ")"
ExpList	→	Exp (, Exp) *

We introduce a start rule, eliminate all left-recursion, and establish precedence.

Grammar rules

The grammar rules directly reflect our BNF!

NB: We add some non-terminals to help our visitors.

```
void Goal() : {} { StmList() <EOF> }
void StmList() : {} { Stm() ( ";" Stm() ) * }

void Stm() : {} { Assignment() | PrintStm() }

/* distinguish reading and writing Id */
void Assignment() : {} { WriteId() ":=" Exp() }
void WriteId() : {} { <IDENTIFIER> }

void PrintStm() : {} { "print" "(" ExpList() ")" }

void ExpList() : {} { Exp() ( AppendExp() ) * }
void AppendExp() : {} { "," Exp() }

void Exp() : {} { MulExp() ( PlusOp() | MinOp() ) * }
void PlusOp() : {} { "+" MulExp() }
void MinOp() : {} { "-" MulExp() }

void MulExp() : {} { PrimExp() ( MulOp() | DivOp() ) * }
void MulOp() : {} { "*" PrimExp() }
void DivOp() : {} { "/" PrimExp() }

void PrimExp() : {} { ReadId() | Num() | StmExp() }
void ReadId() : {} { <IDENTIFIER> }
void Num() : {} { <INTEGER_LITERAL> }
void StmExp() : {} { "(" StmList() "," Exp() ")" }
```

JavaCC rules have the following form:

```
void nonTerminalName() :
{
     /* Java Declarations */
}
{
     /* Rule definition */
}
```

As there are no Java variables that we have to declare for our example grammar, each rule starts with an empty pair of curly braces.

Java Tree Builder

JTB automatically generates actions to build the syntax tree, and visitors to visit it.

original source LOC	441
generated source LOC	4912

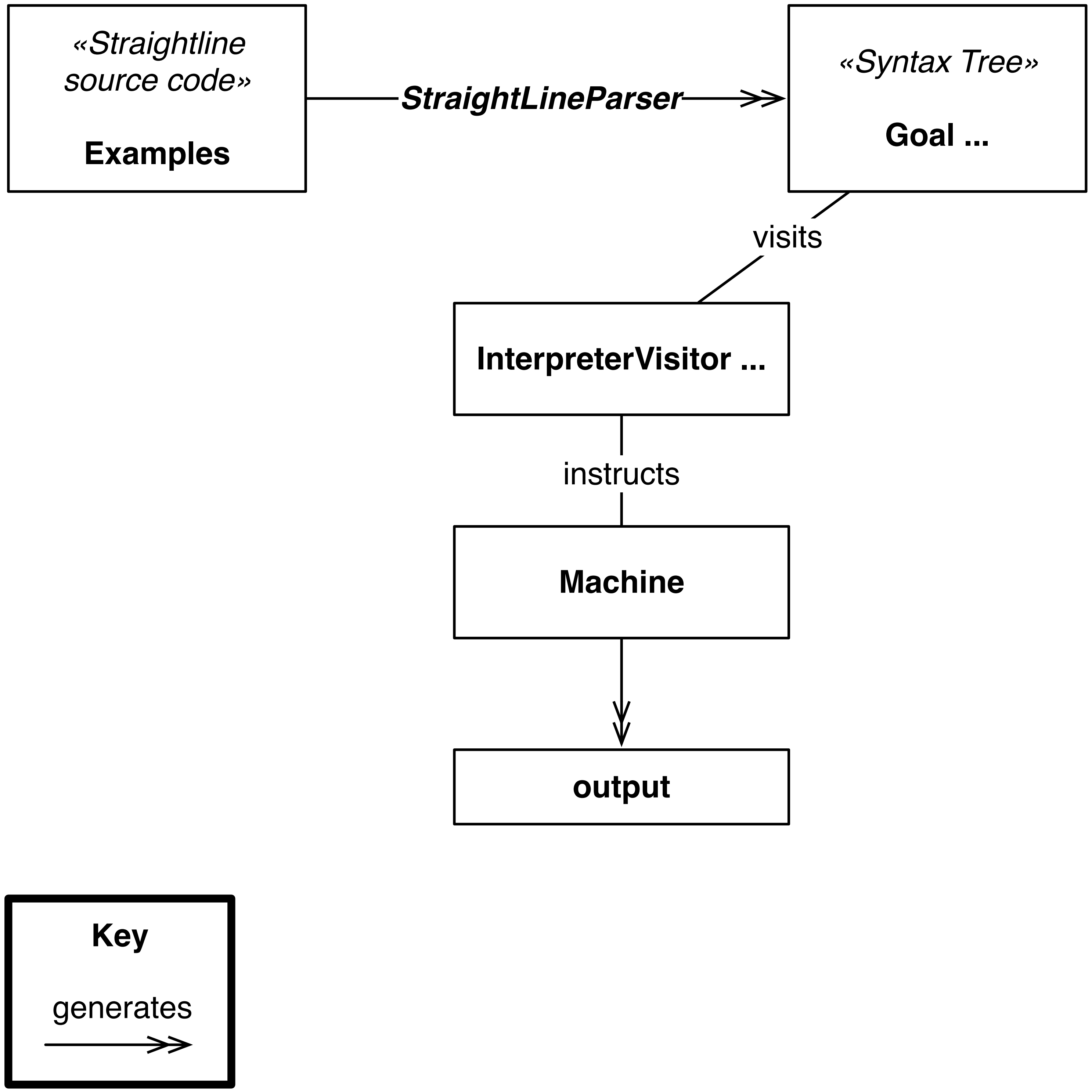
```
// Generated by JTB 1.3.2
options {
   JAVA_UNICODE_ESCAPE = true;
}
PARSER_BEGIN(StraightLineParser)
package parser;
import syntaxtree.*;
import java.util.Vector;

public class StraightLineParser
{
}
…
Goal Goal() :
{
   StmList n0;
   NodeToken n1;
   Token n2;
}
{
   n0=StmList()
   n2=<EOF> {
      n2.beginColumn++; n2.endColumn++;
      n1 = JTBToolkit.makeNodeToken(n2);
   }

   { return new Goal(n0,n1); }
}
...
```

Note how Java declarations are automatically generated for the variables needed to construct the parse tree.

Straightline Interpreter Runtime



At run time, the generated parser builds a parse tree from our straight-line source code, and our interpreter visits the tree to produce output with the help of the abstract machine.

The interpreter

```
package interpreter;
import ...;
public class StraightLineInterpreter {
  Goal parse;
  StraightLineParser parser;

  public static void main(String [] args) {
    System.out.println(new StraightLineInterpreter(System.in).interpret());
  }

  public StraightLineInterpreter(InputStream in) {
    parser = new StraightLineParser(in);
    this.initParse();
  }

  private void initParse() {
    try { parse = parser.Goal(); }
    catch (ParseException e) { ... }
  }

  public String interpret() {
    assert(parse != null);
    Visitor visitor = new Visitor();
    visitor.visit(parse);
    return visitor.result();
  }
}
```

The interpreter simply runs the parser and visits the parse tree.

An abstract machine for straight line code

```
package interpreter;
import java.util.*;
public class Machine {
  private Hashtable<String,Integer> store; // current values of variables
  private StringBuffer output;             // print stream so far
  private int value;                       // result of current expression
  private Vector<Integer> vlist;           // list of expressions computed

  public Machine() {
    store = new Hashtable<String,Integer>();
    output = new StringBuffer();
    setValue(0);
    vlist = new Vector<Integer>();
  }
  void assignValue(String id) { store.put(id, getValue()); }
  void appendExp() { vlist.add(getValue()); }
  void printValues() {...}
  void setValue(int value) {...}
  int getValue() { return value; }
  void readValueFromId(String id) {
    assert isDefined(id); // precondition
    this.setValue(store.get(id));
  }
  private boolean isDefined(String id) { return store.containsKey(id); }
  String result() { return this.output.toString(); }
}
```

The Visitor interacts with this machine as it visits nodes of the program.

Our “abstract machine” simply provides a representation of the abstract state of our straight-line language, and offers operations to manipulate that state.

The `store` keeps track of the current state of all variables, the `output` keeps track of the printed results, `value` represents the result of the current expression, and `vlist` is the list of expressions evaluated thus far.

Operations on the abstract machine allow the values of variables to be updated, values to be printed, and so on.

The visitor

```
package interpreter;
import visitor.DepthFirstVisitor;
import syntaxtree.*;

public class Visitor extends DepthFirstVisitor {
  Machine machine;
  public Visitor() { machine = new Machine(); }
  public String result() { return machine.result(); }

  public void visit(Assignment n) {
    n.f0.accept(this);
    n.f1.accept(this);
    n.f2.accept(this);
    String id = n.f0.f0.tokenImage;
    machine.assignValue(id);
  }
  public void visit(PrintStm n) { ... }
  public void visit(AppendExp n) { ... }
  public void visit(PlusOp n) { ... }
  public void visit(MinOp n) { ... }
  public void visit(MulOp n) { ... }
  public void visit(DivOp n) { ... }
  public void visit(ReadId n) { ... }
  public void visit(Num n) { ... }
}
```

f0 → *WriteId()*
f1 → "`:=`"
f2 → *Exp()*

The Visitor interprets interesting nodes by directly interacting with the abstract machine.

The actual visitor has to interpret each node visited, and perform the corresponding action on the abstract machine. Here we visit an assignment node (i.e., id := exp).

We visit the sub-nodes in a suitable order. The default visitors just store the token names in a way that we can access them. The only visitors that we need to implement are the ones that actually cause side effects to the machine state, i.e., the arithmetic operations of the expressions, variable assignments, and so on.

The sub-node f2 represents the expressions, so visiting it will cause value to be updated. We just need to assign this value to the variable whose name is found in sub-node f0.

See:

http://scg.unibe.ch/download/lectures/cc-examples/cc-StraightLinePL/

```
git clone git://scg.unibe.ch/lectures-cc-examples
```

What you should know!

- ✎ *Why do bottom-up parsers yield rightmost derivations?*
- ✎ *What is a "handle"? How is it used?*
- ✎ *What is "handle-pruning"?How does a shift-reduce parser work?*
- ✎ *When is a grammar LR(k)?*
- ✎ *Which is better for hand-coded parsers, LL(1) or LR(1)?*
- ✎ *What kind of parsers does JavaCC generate?*
- ✎ *How does the Visitor pattern help you to implement parsers?*

Can you answer these questions?

- ✎ *What are “shift-reduce” errors?*
- ✎ *How do you eliminate them?*
- ✎ *Which is more expressive? LL(k) or LR(k)?*
- ✎ *How would you implement the Visitor pattern in a dynamic language (without overloading)?*
- ✎ *How can you manipulate your grammar to simplify your JTB-based visitors?*