UNIVERSITÄT BERN

4. Parsing in Practice

Prof. O. 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.

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.

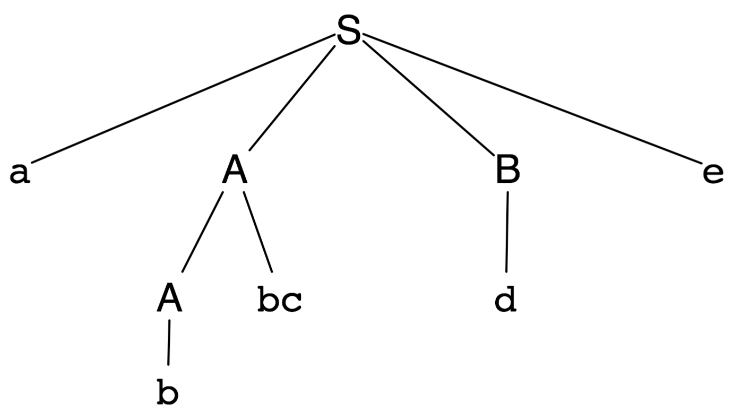
Example

Consider the grammar:

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

and the input string: abbcde

Production	Sentential Form
3	a**b**bcde
2	a**Abc**de
4	aA**d**e
1	**aABe**
	S



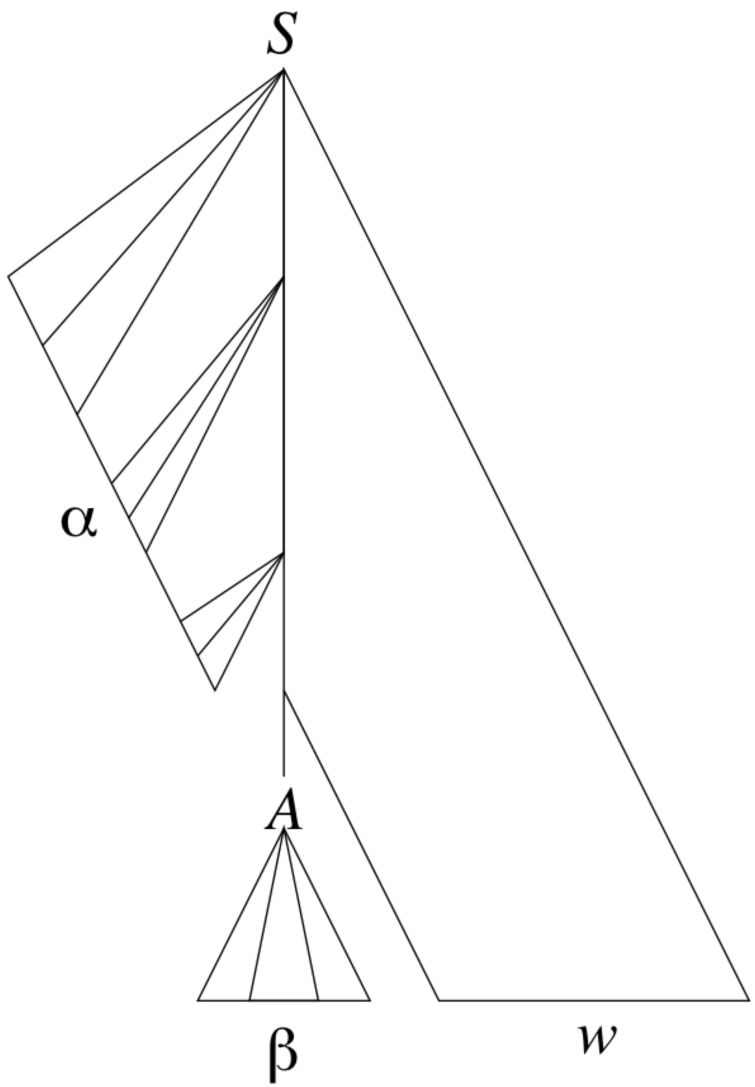
The trick appears to be scanning the input and finding valid sentential forms.

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*.

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 $A \rightarrow \beta$

Example

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	$\underline{\texttt{id}} - \texttt{num} * \texttt{id}$

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 γ_i
2. Replace β_i with A_i to generate γ_{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 $|\beta|$ symbols off the stack
        II. Push A onto the stack

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

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

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

Stack	Input	Action
$	id – num * id	shift
$id	– num * id	reduce 9
$⟨factor⟩	– num * id	reduce 7
$⟨term⟩	– num * id	reduce 4
$⟨expr⟩	– num * id	shift
$⟨expr⟩ –	num * id	shift
$⟨expr⟩ – num	* id	reduce 8
$⟨expr⟩ – ⟨factor⟩	* id	reduce 7
$⟨expr⟩ – ⟨term⟩	* id	shift
$⟨expr⟩ – ⟨term⟩ *	id	shift
$⟨expr⟩ – ⟨term⟩ * id		reduce 9
$⟨expr⟩ – ⟨term⟩ * ⟨factor⟩		reduce 5
$⟨expr⟩ – ⟨term⟩		reduce 3
$⟨expr⟩		reduce 1
$⟨goal⟩		accept

5 shifts + 9 reduces + 1 accept

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

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. $FIRST_k(w) = FIRST_k(y) \Rightarrow \alpha A y = \gamma B x$

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.*

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 $A \rightarrow c$?
  - LR dilemma: pick $A \rightarrow b$ or $B \rightarrow b$?

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).

The JavaCC input format

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

Examples

Token specification:

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

Production:

```
void StatementListReturn() :
{}
{
  ( Statement() )* "return" Expression() ";"
}
```

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.

> *Design Patterns*, 1995, Gamma, Helm, Johnson, Vlissides

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.

Second Approach: Dedicated Methods

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

```
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();
 }
}
```

```
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.

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.

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.
  - Visitors can accumulate state.
- 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 which can be traversed using visitors.

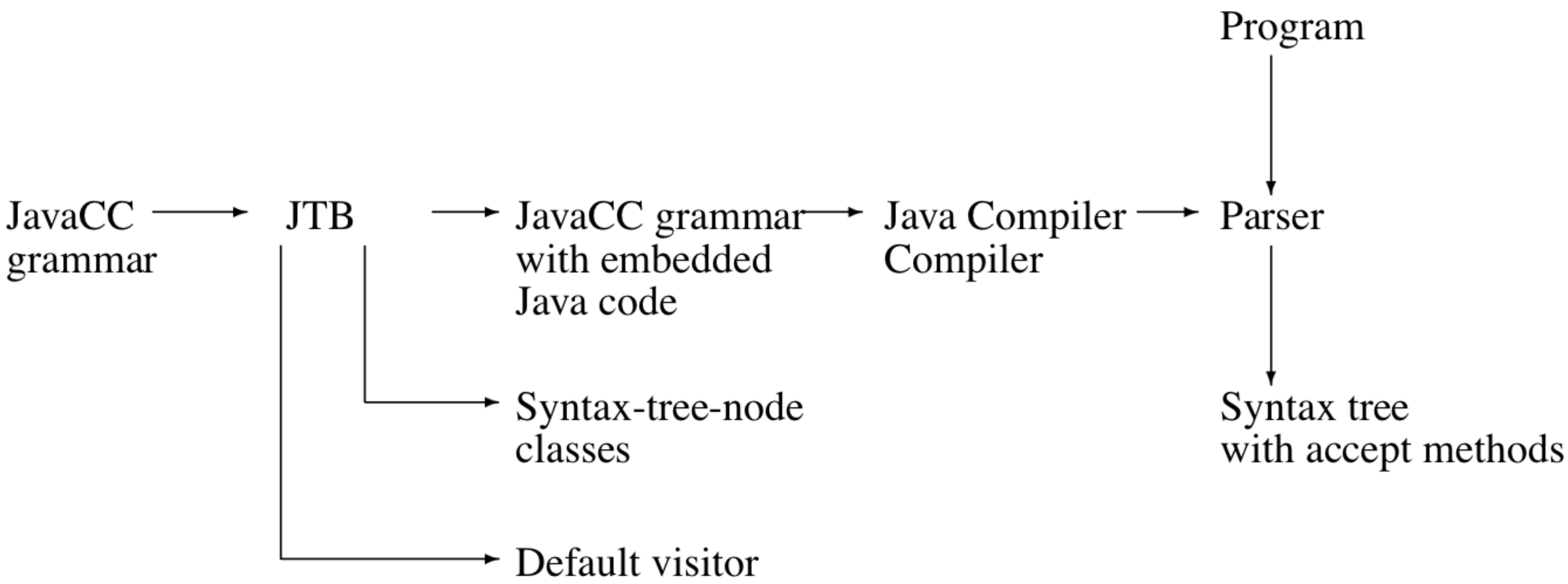
> transforms a bare JavaCC grammar into three components:

- 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*

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: ":=" >
...
}

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

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

more tokens here!

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() ")" }
```

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); }
}
...
```

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.

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.

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?*

License

http://creativecommons.org/licenses/by-sa/3.0/