

Structure of JavaCC File

```
options{
    /* Code to set various options flags */
}

PARSER_BEGIN(MyParser)

public class MyParser {
    /* Java program is placed here */
}

PARSER_END(MyParser)

TOKEN_MGR_DECLS :
{
    /* Declarations used by lexical analyser */
}

/* Token Rules and Actions */

/* JavaCC Rules and Actions - EBNF for language */
```

JavaCC Rules

- JavaCC rules correspond to EBNF rules
- JavaCC rules have the form:

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

- Non-terminals in JavaCC rules have the form of a Java method call
- Terminals in JavaCC rules are between < and >
- For example, the CFG rules:
 $S \rightarrow \text{while } (E) S$
 $S \rightarrow V = E;$
- Would be represented by the JavaCC rule:

```
void statement() : {}  
{  
    <WHILE> <LPAREN> expression() <RPAREN> statement()  
    | variable() <ASSIGN> expression() <SEMICOLON>  
}
```

Example JavaCC File

- We now examine a JavaCC file that parses prefix expressions
- Prefix expression examples:

Prefix expression	Equivalent infix expression
+ 3 4	3 + 4
+ - 2 1 5	(2 - 1) + 5
+ - 3 4 * 5 6	(3 - 4) + (5 * 6)
+ - * 3 4 5 6	((3 * 4) - 5) + 6
+ 3 - 4 * 5 6	3 + (4 - (5 * 6))

- CFG for prefix expressions:

$$E \rightarrow + E E$$

$$E \rightarrow - E E$$

$$E \rightarrow * E E$$

$$E \rightarrow / E E$$

$$E \rightarrow num$$

- In JavaCC:

```
void expression(): {}
{
    <PLUS> expression() expression()
  | <MINUS> expression() expression()
  | <TIMES> expression() expression()
  | <DIVIDE> expression() expression()
  | <INTEGER_LITERAL>
}
```

Lookahead

- Consider the following JavaCC fragment:

```
void S(): {}  
{  
    "a" "b" "c"  
    | "a" "d" "c"  
}
```

- Is this grammar LL(1)?
- A LOOKAHEAD directive, placed at a choice point, will allow JavaCC to use a lookahead > 1 :

```
void S(): {}  
{  
    LOOKAHEAD(2) "a" "b" "c"  
    | "a" "d" "c"  
}
```

- JavaCC will look at the next two symbols, before deciding which rule for S to use

Lookahead

- LOOKAHEAD directives are placed at “choice points” - places in the grammar where there is more than one possible rule that can match.
- Problem grammar:

```
void S(): {}  
{  
    "A" (("B" "C") | ("B" "D"))  
}
```

- Solution using LOOKAHEAD:

```
void S(): {}  
{  
    "A" ( LOOKAHEAD(2) ("B" "C") | ("B" "D"))  
}
```

- Another possible solution?

```
void S(): {}  
{  
    LOOKAHEAD(2) "A" (("B" "C") | ("B" "D"))  
}
```

- If not, why not?

JavaCC and Non-LL(k)

- JavaCC will produce a parser for grammars that are not LL(1) (and even for grammars that are not LL(k), for any k)
- The parser that is produced is not guaranteed to correctly parse the language described by the grammar
- A warning will be issued when JavaCC is run on a non-LL(1) grammar
- For a non-LL(1) grammar, the rule that appears *first* will be used:

```
void S() : {}  
{  
    "a" "b" "c"  
    | "a" "b" "d"  
}
```

JavaCC and Non-LL(k)

- Infamous dangling else.
- Why doesn't the following grammar work?

```
void statement(): {}  
{  
    <IF> expression() <THEN> statement()  
    | <IF> expression() <THEN> statement() <ELSE> statement()  
    | /* Other statement definitions */  
}
```

- The following grammar will work, but a warning will be given by JavaCC:

```
void statement() : {}  
{  
    <IF> expression() <THEN> statement() optionalelse()  
    | /* Other statement definitions */  
}
```

```
void optionalelse() : {}  
{  
    <ELSE> statement()  
    | /* nothing */ {}  
}
```

JavaCC and Non-LL(k)

- What about this grammar?

```
void statement() : {}  
{  
    <IF> expression() <THEN> statement() optionalelse()  
    | /* Other statement definitions */  
}
```

```
void optionalelse() : {}  
{  
    /* nothing */ {}  
    | <ELSE> statement()  
}
```

- Why doesn't this grammar work?
- The following grammar works correctly, but also produces a warning:

```
void statement() : {}  
{  
    <IF> expression() <THEN> statement() (<ELSE> statement)?  
    | /* Other statement definitions */  
}
```

JavaCC and Non-LL(k)

- The following grammar works correctly:

```
void statement() : {}  
{  
    <IF> expression() <THEN> statement()  
    (LOOKAHEAD(1) <ELSE> statement)?  
    | /* Other statement definitions */  
}
```

- This grammar produces no warnings - not because it is any more safe - if you include a LOOKAHEAD directive, JavaCC assumes you know what you are doing.

Specifying the Grammar of the Straight Line Programming Language

As a reminder, the grammar for the straight line programming language is as follows:

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

Specifying the Grammar of the Straight Line Programming Language

```
options { JAVA_UNICODE_ESCAPE = true; }

PARSER_BEGIN(SLPParser)
public class SLPParser {
    public static void main(String args[]) {
        SLPParser parser;
        if (args.length == 0) {
            System.out.println("SLP Parser: Reading from standard input . . .");
            parser = new SLPParser(System.in);
        } else if (args.length == 1) {
            System.out.println("SLP Parser: Reading from file " + args[0] + " . . .");
            try {
                parser = new SLPParser(new java.io.FileInputStream(args[0]));
            } catch (java.io.FileNotFoundException e) {
                System.out.println("SLP Parser: File " + args[0] + " not found.");
                return;
            }
        } else {
            System.out.println("SLP Parser: Usage is one of:");
            System.out.println("        java SLPParser < inputfile");
            System.out.println("OR");
            System.out.println("        java SLPParser inputfile");
            return;
        }
        try {
            parser.Prog();
            System.out.println("SLP Parser: SLP program parsed successfully.");
        } catch (ParseException e) {
            System.out.println(e.getMessage());
            System.out.println("SLP Parser: Encountered errors during parse.");
        }
    }
}
PARSER_END(SLPParser)
```

Specifying the Grammar of the Straight Line Programming Language

```
/*
***** SECTION 3 - TOKEN DEFINITIONS *****
*/

TOKEN_MGR_DECLS :
{
    static int commentNesting = 0;
}

SKIP : /** Ignoring spaces/tabs/newlines **/
{
    " "
    | "\t"
    | "\n"
    | "\r"
    | "\f"
}

SKIP : /* COMMENTS */
{
    "/"* { commentNesting++; } : IN_COMMENT
}

<IN_COMMENT> SKIP :
{
    "/"* { commentNesting++; }
    | "*"/* { commentNesting--;
        if (commentNesting == 0)
            SwitchTo(DEFAULT);
    }
    | <~[]>
}

```

Specifying the Grammar of the Straight Line Programming Language

```
TOKEN : /* Keywords and punctuation */
{
  < SEMIC : ";" >
| < ASSIGN : ":@" >
| < PRINT : "print" >
| < LBR : "(" >
| < RBR : ")" >
| < COMMA : "," >
| < PLUS_SIGN : "+" >
| < MINUS_SIGN : "-" >
| < MULT_SIGN : "*" >
| < DIV_SIGN : "/" >
}
```

```
TOKEN : /* Numbers and identifiers */
{
  < NUM : (<DIGIT>)+ >
| < #DIGIT : ["0" - "9"] >
| < ID : (<LETTER>)+ >
| < #LETTER : ["a" - "z", "A" - "Z"] >
}
```

```
TOKEN : /* Anything not recognised so far */
{
  < OTHER : ~[] >
}
```

Specifying the Grammar of the Straight Line Programming Language

```
/******  
***** SECTION 4 - THE GRAMMAR *****  
*****/  
  
void Prog() : {}  
{  
    Stm() <EOF>  
}  
  
void Stm() : {}  
{  
    (SimpleStm() [<SEMIC> Stm()] )  
}  
  
void SimpleStm() : {}  
{  
    (Ident() <ASSIGN> Exp())  
| (<PRINT> <LBR> ExpList() <RBR>)  
}  
  
void Exp() : {}  
{  
    (SimpleExp() [BinOp() Exp()] )  
}  
  
void SimpleExp() : {}  
{  
    IdExp()  
| NumExp()  
| (<LBR> Stm() <COMMA> Exp() <RBR>)  
}
```

Specifying the Grammar of the Straight Line Programming Language

```
void Ident() : {}
{
    <ID>
}

void IdExp() : {}
{
    <ID>
}

void NumExp() : {}
{
    <NUM>
}

void ExpList() : {}
{
    (Exp() [<COMMA> ExpList()] )
}

void BinOp() : {}
{
    <PLUS_SIGN>
    | <MINUS_SIGN>
    | <MULT_SIGN>
    | <DIV_SIGN>
}
```

JavaCC Actions

- Each JavaCC rule is converted into a parsing method (just like a recursive descent parser created by hand)
- We can add arbitrary Java code to these methods
- We can also add instance variables and helper methods that every parser method can access
- Adding instance variables:

```
PARSER_BEGIN(MyParser)

public class MyParser {

    /* instance variables and helper methods */

    /* Optional 'main' method (the 'main'
    method can also be in a separate file)

    }

PARSER_END(MyParser)
```

JavaCC Rules

```
<return type> <rule name>() :  
{  
    /* local variables */  
}  
{  
    Rule1  
    | Rule2  
    | ...  
}
```

- Each rule can contain arbitrary Java code between { and }
- JavaCC rules can also return values (work just like any other method)
- Use “<variable> =” syntax to obtain values of method calls

JavaCC Rules

- Building A JavaCC Calculator
- How would we change the following JavaCC file so that it computed the value of the expression, as well as parsing the expression?

```
void expression(): {}  
{  
    term() <EOF>  
}
```

```
void term(): {}  
{  
    factor() ((<MULTIPLY> | <DIVIDE>) factor())*  
}
```

```
void factor(): {}  
{  
    <INTEGER_LITERAL>  
    | <MINUS> factor()  
    | <LPAREN> term() <RPAREN>  
}
```

JavaCC Rules

```
int expression():
{ int result; }
{
    result = term() <EOF>
    { return result; }
}

int term():
{Token t; int result; int rhs;}
{
    result = factor() ((t = <MULTIPLY> | t = <DIVIDE>) rhs = factor()
        { if (t.kind == MULTIPLY)
            result = result * rhs;
          else
            result = result / rhs;
        }
        )*
    { return result; }
}

int factor():
{int value; Token t;}
{
    t = <INTEGER_LITERAL>
    { return Integer.parseInt(t.image); }
| <MINUS> value = factor()
    { return 0 - value; }
| <LPAREN> value = term() <RPAREN>
    { return value; }
}
```

Parsing an Expression

- Function to parse a factor is called, result is stored in `result`
- The next token is observed, to see if it is a `*` or `/`.
- If so, function to parse a factor is called again, storing the result in `rhs`. The value of `result` is updated
- The next token is observed to see if it is a `*` or `/`.
- ...
- Some example expressions:
 - 4
 - 3 * 4
 - 1 * 2 / 3

Input Parameters

- JavaCC rules \equiv function calls in generated parser
- JavaCC rules can have *input parameters* as well as return values
- Syntax for rules with parameters is the same as standard method calls
- Consider the following JavaCC file for expressions:

```
void expression(): {}
{
    term() expressionprime()
}

void expressionprime(): {}
{
    <PLUS> term() expressionprime()
    | <MINUS> term() expressionprime()
    | {}
}
```

Input Parameters

- What should `<PLUS> term() expressionprime()` return?
 - Get the value of the previous term
 - Add that value to `term()`
 - Combine the result with whatever `expressionprime()` returns
- How can we get the value of the previous term?
 - Have it passed in as a parameter
- How can we combine the result with whatever `expressionprime()` returns?
 - Pass the result into `expressionprime()`, and have `expressionprime()` do the combination

Input Parameters

```
int expression():
{int firstterm; int result;}
{
    firstterm = term()
    result = expressionprime(firstterm)
    { return result; }
}

int expressionprime(int firstterm):
{ int nextterm; int result; }
{
    <PLUS> nextterm = term()
    result = expressionprime(firstterm + nextterm)
    { return result; }
| <MINUS> nextterm = term()
    result = expressionprime(firstterm - nextterm)
    { return result; }
| { return firstterm; }
}
```

An Interpreter for the Straight Line Programming Language

```
options { JAVA_UNICODE_ESCAPE = true; }

PARSER_BEGIN(SLPInterpreter)

public class SLPInterpreter {
    public static void main(String args[]) {
        SLPInterpreter interpreter;
        if (args.length == 0) {
            System.out.println("SLP Interpreter: Reading from standard input...");
            interpreter = new SLPInterpreter(System.in);
        } else if (args.length == 1) {
            System.out.println("SLP Interpreter: Reading from file " + args[0] + "...");
            try {
                interpreter = new SLPInterpreter(new java.io.FileInputStream(args[0]));
            } catch (java.io.FileNotFoundException e) {
                System.out.println("SLP Interpreter: File " + args[0] + " not found.");
                return;
            }
        } else {
            System.out.println("SLP Interpreter: Usage is one of:");
            System.out.println("        java SLPInterpreter < inputfile");
            System.out.println("OR");
            System.out.println("        java SLPInterpreter inputfile");
            return;
        }
        try {
            interpreter.Prog();
        } catch (ParseException e) {
            System.out.println(e.getMessage());
            System.out.println("SLP Interpreter: Encountered errors during parse.");
        }
    }
}

PARSER_END(SLPInterpreter)
```

An Interpreter for the Straight Line Programming Language

```

/*****
**** SECTION 3 - TOKEN DEFINITIONS ****
*****/

TOKEN_MGR_DECLS :
{
    static int commentNesting = 0;
}

SKIP : /** Ignoring spaces/tabs/newlines **/
{
    " "
    | "\t"
    | "\n"
    | "\r"
    | "\f"
}

SKIP : /* COMMENTS */
{
    "/"* { commentNesting++; } : IN_COMMENT
}

<IN_COMMENT> SKIP :
{
    "/"* { commentNesting++; }
    | "*"/* { commentNesting--;
        if (commentNesting == 0)
            SwitchTo(DEFAULT);
    }
    | <~ []>
}

```

An Interpreter For the Straight Line Programming Language

```
TOKEN : /* Keywords and punctuation */
{
  < SEMIC : ";" >
| < ASSIGN : ":@" >
| < PRINT : "print" >
| < LBR : "(" >
| < RBR : ")" >
| < COMMA : "," >
| < PLUS_SIGN : "+" >
| < MINUS_SIGN : "-" >
| < MULT_SIGN : "*" >
| < DIV_SIGN : "/" >
}
```

```
TOKEN : /* Numbers and identifiers */
{
  < NUM : (<DIGIT>)+ >
| < #DIGIT : ["0" - "9"] >
| < ID : (<LETTER>)+ >
| < #LETTER : ["a" - "z", "A" - "Z"] >
}
```

```
TOKEN : /* Anything not recognised so far */
{
  < OTHER : ~[] >
}
```

An Interpreter for the Straight Line Programming Language

```
/******  
***** SECTION 4 - THE GRAMMAR *****  
*****/  
  
void Prog() :  
{Table t;}  
{  
    t=Stm(null) <EOF>  
}  
  
Table Stm(Table t) :  
{}  
{  
    (t=SimpleStm(t) [<SEMIC> t=Stm(t)] ) {return t;}  
}  
  
Table SimpleStm(Table t) :  
{String id; IntAndTable it; IntListAndTable ilt;}  
{  
    (id=Ident() <ASSIGN> it=Exp(t))  
    {  
        if (t == null)  
            return new Table(id,it.i,t);  
        else  
            return t.update(t,id,it.i);  
    }  
| (<PRINT> <LBR> ilt=ExpList(t) <RBR>)  
    {  
        ilt.il.print();  
        return ilt.t;  
    }  
}  
}
```

An Interpreter for the Straight Line Programming Language

```
IntAndTable Exp(Table t) :
{IntAndTable arg1, arg2; int oper;}
{
    (arg1=SimpleExp(t)
    [oper=BinOp() arg2=Exp(arg1.t)
    { switch(oper) {
        case 1: return new IntAndTable(arg1.i+arg2.i,arg2.t);
        case 2: return new IntAndTable(arg1.i-arg2.i,arg2.t);
        case 3: return new IntAndTable(arg1.i*arg2.i,arg2.t);
        case 4: return new IntAndTable(arg1.i/arg2.i,arg2.t);
    }
    ]
    )
    {return arg1;}
}
```

```
IntAndTable SimpleExp(Table t) :
{IntAndTable it;}
{
    it=IdExp(t) {return it;}
| it=NumExp(t) {return it;}
| (<LBR> t=Stm(t) <COMMA> it=Exp(t) <RBR>) {return it;}
}
```

```
String Ident() :
{Token tok;}
{
    tok=<ID> {return tok.image;}
}
```

An Interpreter for the Straight Line Programming Language

```
IntAndTable IdExp(Table t) :
{Token tok;}
{
    tok=<ID> {return new IntAndTable(t.lookup(t,tok.image),t);}
}

IntAndTable NumExp(Table t) :
{Token tok;}
{
    tok=<NUM> {return new IntAndTable(Integer.parseInt(tok.image),t);}
}

IntListAndTable ExpList(Table t) :
{IntAndTable it;IntListAndTable ilt;}
{
    (it=Exp(t)
     [<COMMA> ilt=ExpList(it.t)
     {return new IntListAndTable(new IntList(it.i,ilt.il),ilt.t);}
     ])
    {return new IntListAndTable(new IntList(it.i,null),it.t);}
}

int BinOp() : {}
{
    <PLUS_SIGN> {return 1;}
| <MINUS_SIGN> {return 2;}
| <MULT_SIGN> {return 3;}
| <DIV_SIGN> {return 4;}
}
```

Building ASTs With JavaCC

- Instead of returning values, return trees
- Call constructors to build subtrees
- Combine subtrees into larger trees

```
ASTExpression term():
{Token t; ASTExpression result; ASTExpression rhs;}
{
    result = factor() ((t = <MULTIPLY> | t = <DIVIDE>) rhs = factor()
        { result = new ASTOperatorExpression(result,rhs,t.image); }
        )*
    { return result; }
}
```

```
ASTExpression factor():
{ASTExpression value; Token t;}
{
    t = <INTEGER_LITERAL>
        { return new ASTIntegerLiteral(Integer.parseInt(t.image)); }
    | <MINUS> value = factor()
        { return new ASTOperatorExpression(new ASTIntegerLiteral(0),
            value,ASTOperatorExpression.MINUS);}
    | <LPAREN> value = term() <RPAREN>
        { return value; }
}
```

A Syntax Tree Builder for the Straight Line Programming Language

PARSER_BEGIN(SLPStringBuilder)

```
public class SLPStringBuilder {

    public static void main(String args[]) {
        SLPStringBuilder treebuilder;
        if (args.length == 0) {
            System.out.println("SLP Tree Builder: Reading from standard input . . .");
            treebuilder = new SLPStringBuilder(System.in);
        } else if (args.length == 1) {
            System.out.println("SLP Tree Builder: Reading from file " + args[0] + " . . .")
            try {
                treebuilder = new SLPStringBuilder(new java.io.FileInputStream(args[0]));
            } catch (java.io.FileNotFoundException e) {
                System.out.println("SLP Tree Builder: File " + args[0] + " not found.");
                return;
            }
        } else {
            System.out.println("SLP Tree Builder: Usage is one of:");
            System.out.println("        java SLPStringBuilder < inputfile");
            System.out.println("OR");
            System.out.println("        java SLPStringBuilder inputfile");
            return;
        }
        try {
            Stm s = treebuilder.Prog();
            s.interp();
        } catch (ParseException e) {
            System.out.println(e.getMessage());
            System.out.println("SLP Tree Builder: Encountered errors during parse.");
        }
    }
}
```

PARSER_END(SLPStringBuilder)

A Syntax Tree Builder for the Straight Line Programming Language

```

/*****
**** SECTION 3 - TOKEN DEFINITIONS ****
*****/

TOKEN_MGR_DECLS :
{
    static int commentNesting = 0;
}

SKIP : /** Ignoring spaces/tabs/newlines **/
{
    " "
    | "\t"
    | "\n"
    | "\r"
    | "\f"
}

SKIP : /* COMMENTS */
{
    "/"* { commentNesting++; } : IN_COMMENT
}

<IN_COMMENT> SKIP :
{
    "/"* { commentNesting++; }
    | "*"/* { commentNesting--;
        if (commentNesting == 0)
            SwitchTo(DEFAULT);
    }
    | <~[]>
}

```

An Interpreter For the Straight Line Programming Language

```
TOKEN : /* Keywords and punctuation */
{
  < SEMIC : ";" >
| < ASSIGN : ":@" >
| < PRINT : "print" >
| < LBR : "(" >
| < RBR : ")" >
| < COMMA : "," >
| < PLUS_SIGN : "+" >
| < MINUS_SIGN : "-" >
| < MULT_SIGN : "*" >
| < DIV_SIGN : "/" >
}
```

```
TOKEN : /* Numbers and identifiers */
{
  < NUM : (<DIGIT>)+ >
| < #DIGIT : ["0" - "9"] >
| < ID : (<LETTER>)+ >
| < #LETTER : ["a" - "z", "A" - "Z"] >
}
```

```
TOKEN : /* Anything not recognised so far */
{
  < OTHER : ~[] >
}
```

A Syntax Tree Builder for the Straight Line Programming Language

```

Stm Prog() :
{ Stm s; }
{
  s=Stm() <EOF>
  { return s; }
}

Stm Stm() :
{ Stm s1,s2; }
{
  (s1=SimpleStm() [<SEMIC> s2=Stm() {return new CompoundStm(s1,s2);} ] )
  { return s1; }
}

Stm SimpleStm() :
{ String s; Exp e; ExpList el; }
{
  (s=Ident() <ASSIGN> e=Exp()          { return new AssignStm(s,e); }
| (<PRINT> <LBR> el=ExpList() <RBR>)  { return new PrintStm(el); }
}

Exp Exp() :
{ Exp e1,e2; int o; }
{
  (e1=SimpleExp() [o=BinOp() e2=Exp() { return new OpExp(e1,o,e2); } ] )
  { return e1; }
}

Exp SimpleExp() :
{ Stm s; Exp e; }
{
  e=IdExp() { return e; }
| e=NumExp() { return e; }
| (<LBR> s=Stm() <COMMA> e=Exp() <RBR>) { return new EseqExp(s,e); }
}

```

A Syntax Tree Builder for the Straight Line Programming Language

```
String Ident() :
{ Token t; }
{
  t=<ID> { return t.image; }
}

IdExp IdExp() :
{ Token t; }
{
  t=<ID> { return new IdExp(t.image); }
}

NumExp NumExp() :
{ Token t; }
{
  t=<NUM> { return new NumExp(Integer.parseInt(t.image)); }
}

ExpList ExpList() :
{ Exp e; ExpList el; }
{
  (e=Exp() [<COMMA> el=ExpList() { return new PairExpList(e,el); } ] )
  { return new LastExpList(e); }
}

int BinOp() : {}
{
  <PLUS_SIGN> { return 1; }
| <MINUS_SIGN> { return 2; }
| <MULT_SIGN> { return 3; }
| <DIV_SIGN> { return 4; }
}
```