CA4003 - Compiler Construction

JavaCC

David Sinclair

JavaCC

JavaCC is a LL(1) compiler generator.

```
javacc_input ::= javacc_options
                 "PARSER_BEGIN" "(" <IDENTIFIER> ")"
                 java_compilation_unit
                 "PARSER_END" "(" <IDENTIFIER> ")"
                 (production)*
                 <EOF>
```

The Straight Line Programming Language Again

Let's revisit the straight-line programming language from Appel's book (Chapter 1).

Stm	$\longrightarrow$	*Stm* ; *Stm*	(CompoundStm)
Stm	$\longrightarrow$	id := *Exp*	(AssignStm)
Stm	$\longrightarrow$	print (*ExpList*)	(PrintStm)
Exp	$\longrightarrow$	id	(IdExp)
Exp	$\longrightarrow$	num	(NumExp)
Exp	$\longrightarrow$	*Exp Binop Exp*	(OpExp)
Exp	$\longrightarrow$	(*Stm* , *Exp*)	(EseqExp)
ExpList	$\longrightarrow$	*Exp* , *ExpList*	(PairExpList)
ExpList	$\longrightarrow$	*Exp*	(LastExpList)
Binop	$\longrightarrow$	+	(Plus)
Binop	$\longrightarrow$	-	(Minus)
Binop	$\longrightarrow$	$\times$	(Times)
Binop	$\longrightarrow$	/	(Div)

The Straight Line Programming Language Again [2]

This is the lexical analyser from earlier.

```
/*******************************
 ***** SECTION 1 - OPTIONS *****
 *******************************/

options { JAVA_UNICODE_ESCAPE = true; }

/*********************************
 ***** SECTION 2 - USER CODE *****
 *********************************/

PARSER_BEGIN(SLPTokeniser)

public class SLPTokeniser {

  public static void main(String args[]) {

    SLPTokeniser tokeniser;
    if (args.length == 0) {
      System.out.println("Reading from standard input . . .");
      tokeniser = new SLPTokeniser(System.in);
    } else if (args.length == 1) {
      try {
        tokeniser = new SLPTokeniser(new java.io.FileInputStream(args[0]));
      } catch (java.io.FileNotFoundException e) {
        System.err.println("File " + args[0] + " not found.");
        return;
      }
    }
```

The Straight Line Programming Language Again [3]

```
    else {
      System.out.println("SLP Tokeniser:  Usage is one of:");
      System.out.println("         java SLPTokeniser < inputfile");
      System.out.println("OR");
      System.out.println("         java SLPTokeniser inputfile");
      return;
    }

    /*
     * We've now initialised the tokeniser to read from the appropriate place,
     * so just keep reading tokens and printing them until we hit EOF
     */

    for (Token t = getNextToken(); t.kind!=EOF; t = getNextToken()) {
      // Print out the actual text for the constants, identifiers etc.
      if (t.kind==NUM)
      {
         System.out.print("Number");
         System.out.print("("+t.image+") ");
      }
      else if (t.kind==ID)
      {
         System.out.print("Identifier");
         System.out.print("("+t.image+") ");
      }
      else
         System.out.print(t.image+" ");
    }
  }
}
PARSER_END(SLPTokeniser)
```


The Straight Line Programming Language Again [4]

```
/*****************************************
 ***** 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);
         }
  | <~[]>
}
```

The Straight Line Programming Language Again [5]

```
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 : ~[] >
}

/*****************************************************************************
 * SECTION 4 - THE GRAMMAR & PRODUCTION RULES - WOULD NORMALLY START HERE *
 *****************************************************************************/
```


Specifying the Grammar

```
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;
    }
```

Specifying the Grammar [2]

```
    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)

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

TOKEN_MGR_DECLS :
{
   static int commentNesting = 0;
}

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


Specifying the Grammar [3]

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

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

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

Specifying the Grammar [4]

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


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

/***********************************
 ***** SECTION 4 - THE GRAMMAR *****
 ***********************************/

void Prog() : {}
{
  Stm() <EOF>
}

void Stm() : {}
{
  (SimpleStm() [<SEMIC> Stm()] )
}
```


Specifying the Grammar [5]

```
void SimpleStm() : {}
{
  (Ident() <ASSIGN> Exp())
| (<PRINT> <LBR> ExpList() <RBR>)
}

void Exp() : {}
{
  (SimpleExp() [BinOp() Exp()]  )
}

void SimpleExp() : {}
{
  IdExp()
| NumExp()
| (<LBR> Stm() <COMMA> Exp() <RBR>)
}

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

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

Specifying the Grammar [6]

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

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

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


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;
    }
```

An Interpreter for the Straight Line Programming Language [2]

```
    try {
      interpreter.Prog();
    } catch (ParseException e) {
      System.out.println(e.getMessage());
      System.out.println("SLP Interpreter:  Encountered errors during parse.");
    }
  }
}
PARSER_END(SLPInterpreter)

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

TOKEN_MGR_DECLS :
{
   static int commentNesting = 0;
}

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


An Interpreter For the Straight Line Programming Language [3]

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

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

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

An Interpreter for the Straight Line Programming Language [4]

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


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

/***********************************
 ***** SECTION 4 - THE GRAMMAR *****
 ***********************************/

void Prog() :
{Table t;}
{
  t=Stm(null) <EOF>
}
```


An Interpreter for the Straight Line Programming Language [5]

```
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 [6]

```
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;}
}
```


An Interpreter for the Straight Line Programming Language [7]

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

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);}
}
```

An Interpreter for the Straight Line Programming Language [8]

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


A Syntax Tree Builder for the Straight Line Programming Language

```
PARSER_BEGIN(SLPTreeBuilder)

public class SLPTreeBuilder {

  public static void main(String args[]) {
    SLPTreeBuilder treebuilder;
    if (args.length == 0) {
      System.out.println("SLP Tree Builder:  Reading from standard input . . .");
      treebuilder = new SLPTreeBuilder(System.in);
    } else if (args.length == 1) {
      System.out.println("SLP Tree Builder:  Reading from file " + args[0] + " . . .");
      try {
        treebuilder = new SLPTreeBuilder(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 SLPTreeBuilder < inputfile");
      System.out.println("OR");
      System.out.println("         java SLPTreeBuilder inputfile");
      return;
    }
```

A Syntax Tree Builder for the Straight Line Programming Language [2]

```
    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(SLPTreeBuilder)

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

TOKEN_MGR_DECLS :
{
   static int commentNesting = 0;
}

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


A Syntax Tree Builder for the Straight Line Programming Language [3]

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

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

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

A Syntax Tree Builder for the Straight Line Programming Language [4]

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


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

/***********************************
 ***** SECTION 4 - THE GRAMMAR *****
 ***********************************/

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


A Syntax Tree Builder for the Straight Line Programming Language [5]

```
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 [6]

```
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); }
}
```


A Syntax Tree Builder for the Straight Line Programming Language [7]

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