Chapter 13

JavaCC

Token manager generator

Parser generator

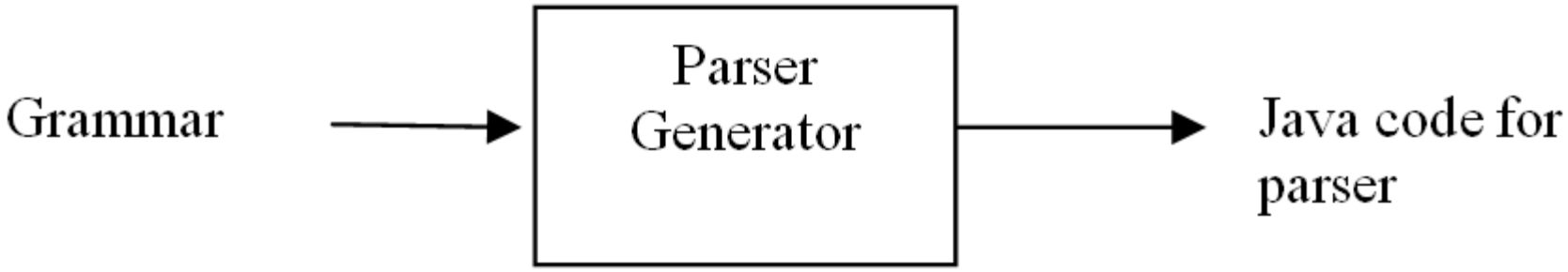
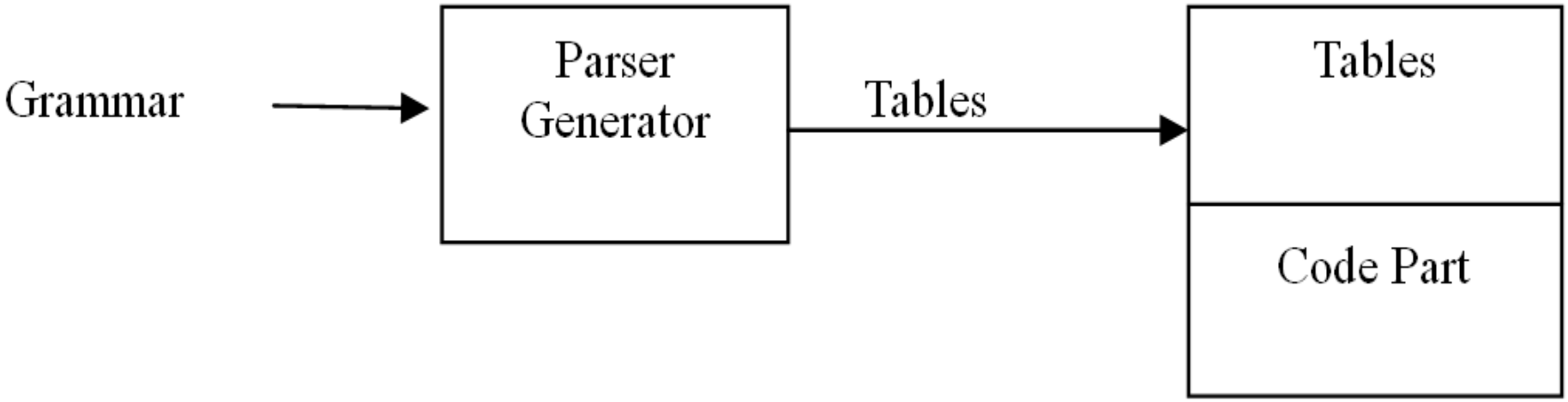


Table-driven parser generator



JavaCC inputs

JavaCC regular expressions

```
"b"|"c"
("b")*              use parens
["b", "c", "d"]
["A"-"Z", "a"-"z"]
["b", "cd", "\n"]   "cd" illegal inside
~["b"]              don't use parens
```

JavaCC input file

```
options
{
    // JavaCC options go here.
}

PARSER_BEGIN(name of parser class)

    // The parser class and other classes go here.

PARSER_END(name of parser class)

TOKEN_MGR_DECLS:
{
    // Declarations of variables and methods for use
    // by the token manager go here.
}

SKIP:
{
    // Regular expressions that describe tokens that
    // the token manager should not pass to the parser
    // go here.
}

TOKEN:
{
    // Regular expressions that describe tokens that
    // the token manager should pass to the parser go here.
}

    // Translation grammar goes here.
```

Format of S1j.jj

```
PARSER_BEGIN(S1j)
import java.io.*;
import java.util.ArrayList;
class S1j
{
   public static void main(String[] arg)
   {
      ...
   }
}
//=========================================
class S1jSymTab
{
   ...
}
//=========================================
class S1jCodeGen
{
   ...
}
PARSER_END(S1j)
```

Matching tokens

1. Always use the longest match possible. For example, suppose a `TOKEN` block is

```
TOKEN:
{
      <T1: ("b")+>
}
```

and the input to the token manager is `"bbbccc"`. Then `<T1>` matches three substrings of the input: the first `"b"`, the first two `b`'s. and all three `b`'s. The token manager in this case returns the longest matched token, which is `"bbb"`.

2. If more than one expression provides the longest match, then use the one listed the first. We have already seen this rule in action with `"println"` and the expression for `ID`. Both match `"println"`. So the order in which we list them in the `TOKEN` block determines which one is used when `"println"` appears in the input stream.

Typical JavaCC input file

```
options
{
   STATIC = false;
   COMMON_TOKEN_ACTION = true;
   // other options can go here
}
//=============================================================
PARSER_BEGIN (name of parser class)

   // The parser class and other classes go here    .

PARSER_END (name of parser class)
//=============================================================
TOKEN_MGR_DECLS:
{
   void CommonTokenAction(Token t)
   {
      System.out.println("image is " + t.image);
   }
}
//=============================================================
SKIP:
{
   " "
 |
   "\n"
 |
   "\r"
 |
   "\t"
}
```

Token block

```
TOKEN:
{
   <PRINTLN: "println">
 |
   <UNSIGNED: (["0"-"9"])+>
 |
   <ID: ["A"-"Z","a"-"z"] (["A"-"Z","a"-"z","0"-"9"])*>
 |
   <ASSIGN: "=">
 |
   <SEMICOLON: ";">
 |
   <LEFTPAREN: "(">
 |
   <RIGHTPAREN: ")">
 |
   <PLUS: "+">
 |
   <MINUS: "-">
 |
   <TIMES: "*">
 |
   <ERROR: ~[]>
}

// Translation grammar goes here ============================
```

S1j.jj

S1j.txt

Files produced by JavaCC

```
S1j.java
ParseException.java
Token.java
S1jConstants.java
S1jTokenManager.java
TokenMgrError.java
SimpleCharStream.java
```

S1jConstants.java

S1jConstants.txt

Using JavaCC

```
javacc S1j.jj
javac S1j.java
java S1j S1
a S1.a
e S1 /c
```

Using star operator

```
void expr(): {Token t;}
{
    term()
    (
        (t="+"|t="-")          // save operator in t
        term()
        {
            if (t.kind == PLUS)
                codeGen.emitInstruction("add" );
            else
                codeGen.emitInstruction("sub");
        }
     )*
}
```

Choice point

Point at which a LOOKAHEAD directive can be placed.

Choice points for various structures

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

    (
        E()
    )?

    (
        F()
    )*

    (
        G()
    )+
}
```

choice point for first " | "

choice point for second " | "

choice point for "?"

choice point for "*"

choice point for "+"

Don't have to compute FOLLOW sets

Suppose T can generate null string:

P -> Q
P -> R
P -> T

Standard P method

```
void P()
{
    if (current token in FIRST(Q))
        q();
    else
    if (current token in FIRST(R))
        R();
    else
    if (current token in FIRST(T)|FOLLOW(P))
        T();
    else
        throw exception
}
```

Alternate P method

```
void P()
{
    if (current token in FIRST(Q))
        Q();
    else
    if (current token in FIRST(R))
        R();
    else        // no selection set test
        T();    // this is the default alternative
}
```

Lambda production applied unconditionally

```
 1 // Fig. 13.19
 2 options
 3 {
 4    STATIC = false;
 5 }
 6 //=================================================
 7 PARSER_BEGIN(Fig1319)
 8 import java.io.*;
 9 class Fig1319
10 {
11   public static void main(String[] args) throws
12                               IOException, ParseException
13   {
14     Fig1319 parser =
15               new Fig1319(new FileInputStream(args[0]));
16     parser.S();
17   }
18 }
19 PARSER_END(Fig1319)
20 //=================================================
21 SKIP:
22 {
23    " "|"\n"|"\r"|"\t"
24 }
25 //=================================================
26 void S(): {}
27 {
28    {System.out.println("hello");}  // Sel set is {<EOF>}
29  |
30    "b" S()                        // Sel set is {"b"}}
31 }
```

LOOKAHEAD directive

```
PARSER_BEGIN(G1304)
class G1304
{
}
PARSER_END(G1304)
void S(): {}
{
   LOOKAHEAD(2)       // Put LOOKAHEAD(2) at choice point
   "b" "c" "d"
|
   "b" "e" "f"
}
```

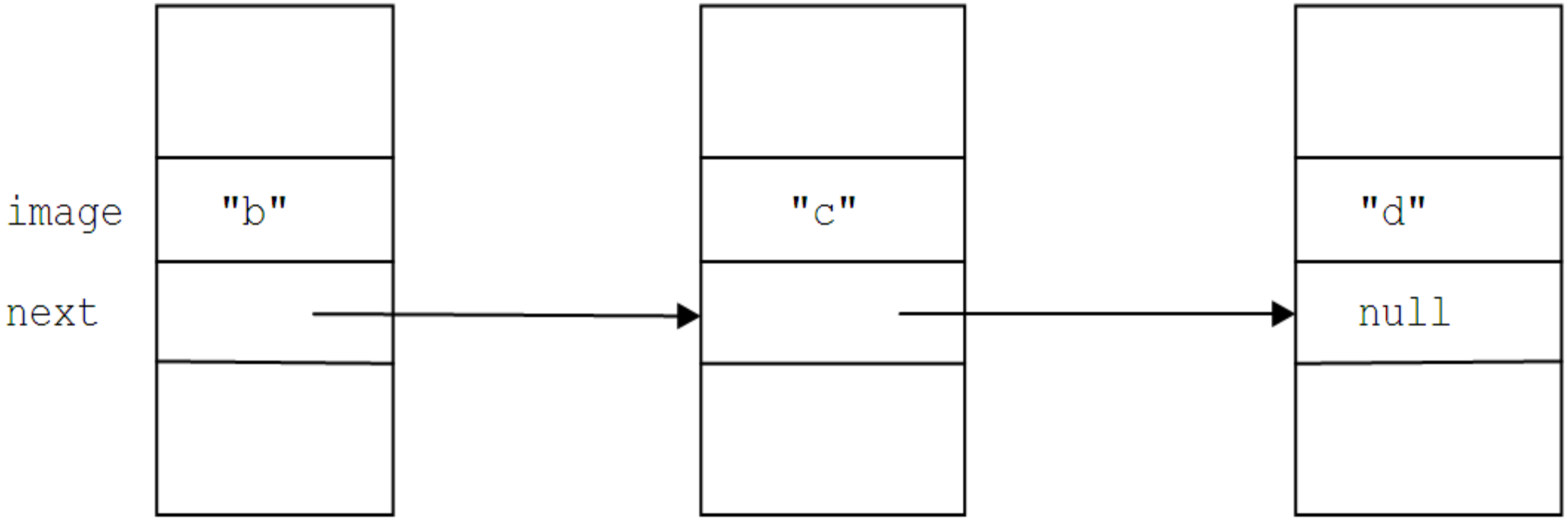
Syntactic lookahead

```
 1 void S(): {}
 2 {
 3    LOOKAHEAD(D()"b")
 4    B()
 5  |
 6    C()
 7 }
 8 //-------------------------------------
 9 void B(): {}
10 {
11    D()
12    "b"
13    "b"
14 }
15 //-------------------------------------
16 void C(): {}
17 {
18    D()
19    "c"
20    "c"
21 }
22 //-------------------------------------
23 void D(): {}
24 {
25    "d"
26    "d"
27 }
```

Semantic lookahead

```
 1 void S(): {}
 2 {
 3  LOOKAHEAD({getToken(1).kind==UNSIGNED &&
 4                               getToken(2).kind==ID})
 5    E()
 6  |
 7    F()
 8 }
 9 //-------------------------------------
 0 void E(): {}
11 {
12    <UNSIGNED>
13    <ID>
14 }
15 //-------------------------------------
16 void F(): {}
17 {
18    <UNSIGNED>
19    <UNSIGNED>
20 }
```

Using token chain



Modified statement() method

```
 1 // second makeComment method
 2 public void makeComment(Token t1, Token t2)
 3 {
 4   outFile.print("; "); // start comment
 5   while (t1 != t2)
 6   {
 7     outFile.print(t1.image + " ");
 8     t1 = t1.next;
 9   }
10   outFile.println();   // terminate comment
11 }
12  //================================================
13 // third makeComment method
14 public void makeComment(String s)
15 {
16   outFile.println("; " + s);
17 }
18 //================================================
19 // modified statement() in the translation grammar
20 void statement(): {Token t; boolean outComment;}
21 {
22    {t = getToken(1);}     // save current token
23    {outComment = true;}
24    (
25       assignmentStatement()
26     |
27       printlnStatement()
28    )
29    // output tokens from t to current token
30    {if (outComment) makeComment(t, getToken(1));}
31 }
```