City University London

IN2009

Language Processors

Week 4

Parsing with JavaCC

Igor Siveroni

Session Plan

Session 4: Parsing with JavaCC

- EBNF
- Parsing with JavaCC
- Lookahead
- JavaCC grammars, semantic actions and values
- Simple expression evaluator

Extended BNF (EBNF)

A few additional operators to shorten definitions:

- e_1 | e_2 | e_3 | ... : choice of e_1, e_2, e_3, etc
- (…) bracketing allowed
- [...] : the expression in […] may be omitted
  [M] = (M)?
- (e)+ : One or more occurrences of e
- (e)* : Zero or more occurrences of e

Extended BNF (EBNF)

Note that these may be nested within each other, so we can have:

((e_1 | e_2)* [e_3]) | e_4

Examples:

IfStatement → **if “(“** Expression **“)”** StatementBlock
[**else** StatementBlock]

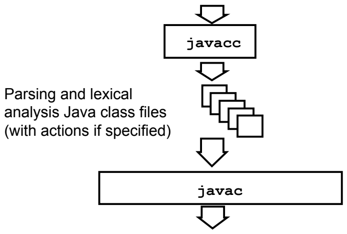
StatementBlock → **“{“** (Statement)+ **“}”**

Expression grammar in EBNF

E → E + T E → E - T E → T	T → T * F T → T / F T → F	F → id F → num F → “(“ E ”)”	*Original*
E → T E’ E’ → + T E’ E’ → - T E’ E’ →	T → F T’ T’ → * F T’ T’ → / F T’ T’ →	F → id F → num F → “(“ E ”)”	*Left-recursion eliminated*
E → T (+ T | - T)*	T → F (* T | / T)*	F → id F → num F → “(“ E ”)”	*EBNF*

JavaCC

JavaCC specification
(includes both token specifications ***and*** grammar, possibly with actions)


javacc
Parsing and lexical analysis Java class files (with actions if specified)
javac

combined token matcher and parser (executing actions if specified)

JavaCC

- **JavaCC is a *parser generator***. Given as input a set of token definitions, a programming language syntax grammar, and a set of actions written in Java, it produces a Java program which will perform lexical analysis to find tokens and then parse the tokens according to the grammar and execute the actions as appropriate.

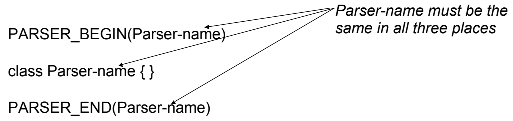
JavaCC

- it works on LL(1) grammars (no need to understand this definition), which are similar to those that recursive descent works for.
- it requires a non-ambiguous grammar with left-recursion removed, so we use the techniques from earlier this session.

For the record: LL(1) grammar

Left-to-right parse, *leftmost* derivation, *1* symbol lookahead

JavaCC input file format (.jj)



Parser-name must be the same in all three places

```
PARSER_BEGIN(Parser-name)

class Parser-name { }

PARSER_END(Parser-name)

/* Lexical items (i.e. token definitions) – see previous examples */

Token-definitions

/* Grammar rules – in a stylised form of EBNF (see next slide). */

Syntax-definitions
```

JavaCC Syntax-definitions

A BNF production: *Non-Terminal-Name -> Right-Hand-Side* is written:

java_return_type *Non-Terminal-Name* (java_parameter_list) : (1)
{ java_block } (2)
{ expansion_choices } (3)

(1) gives the name of the non-terminal being defined

The rest of (1) looks like a Java method declaration. Using this feature we can cause values to be passed up and down the parse tree while the parse takes place (up via return values and down via parameters).

(2) java_block: introduces some Java code which is usually used to declare variables for use in the production

(3) is the EBNF definition and actions…see next slide

JavaCC EBNF expansion_choices

expansion | expansion | ...	where the \`|' separates alternatives.
expansion expansion ...	matches first expansion then second and so on
(expansion_choices)*	matches zero or more expansion_choices
(expansion_choices)+	matches one or more expansion_choices
(expansion_choices)?	matches expansion_choices or empty string
[expansion_choices]	ditto (ie same as ?)
regexp	matches the token matched by the regexp
java_id = regexp	ditto, assigning token to java_id
non-terminal-name (…)	matches the non-terminal
java_id = non-terminal-name (…)	ditto, assigning returned value to java_id

The java_id will usually be declared in the java_block.

Any of these expansions may be followed by some Java code written in {...} and this code (often called an action) will be **executed** when the generated parser matches the expansion.

JavaCC EBNF example

E → T (+ T | - T)*

T → F (* T | / T)*

F → num
F → (E)

```
void E() :
{}
{
  T() ( "+" T() | "-" T() )*
}
```

```
void T() :
{}
{
  F() ( "*" F() | "/" F() )*
}
```

```
void F() :
{}
{
    <NUM>
  | "(" E() ")"
}
```

```
TOKEN :
{
  < NUM: (["0"-"9"])+ >
}
```

JavaCC example: Exp.jj file

```
PARSER_BEGIN(Exp)

public class Exp {
}

PARSER_END(Exp)

SKIP :
{
  " " | "\t" | "\r"
}

TOKEN :
{
  < NUM: (["0"-"9"])+ > | < EOL: "\n" >
}

void S() :
{}
{
    E() <EOL>
  | <EOL>
  | <EOF>
}
```

```
void E() :
{}
{
  T() ( "+" T() | "-" T() )*
}

void T() :
{}
{
  F() ( "*" F() | "/" F() )*
}

void F() :
{}
{
    <NUM>
  | "(" E() ")"
}
```

JavaCC example: `Main.java` file

Another class can instantiate the Exp object and call the S() method:

```
public class Main {
  public static void main(String args[]) throws ParseException {
    Exp parser = new Exp(System.in);
    try {
      System.out.println("Type in an expression on a single line.");
      parser.S();
      System.out.println("Expression parser - parse successful");
    } catch (ParseException e) {
      System.out.println("Expression parser - error in parse");
    }
  }
}
```

JavaCC BNF example

E → T E'

E' → + T E'
E' → - T E'
E' →

T → F T'

T' → * F T'
T' → / F T'
T' →

F → num
F → (E)

```
void E() :
{}
{
  T() Eprime()
}

void Eprime() :
{}
{
    ( "+" T() Eprime() )
  | ( "-" T() Eprime() )
  | {} /* empty */
}
```

```
void T() :
{}
{
  F() Tprime()
}

void Tprime() :
{}
{
    ( "*" F() Tprime() )
  | ( "/" F() Tprime() )
  | {} /* empty */
}
```

```
void F() :
{}
{
    <NUM>
  | "(" E() ")"
}
```

```
TOKEN :
{
  < NUM: (["0"-"9"])+ >
}
```

Local Lookahead (using JavaCC)

Statement	→ { *Statement* * }	BlockStm
Statement	→ *id* **=** *Exp* **;**	AssignmentStm
Statement	→ *id* **[** *Exp* **]** **=** *Exp* **;**	ArrayAssignmentStm

Given token <ID>, the parser needs to know which rule to apply. I needs to lookahead the next token:

```
void Statement() :
{}
{
  BlockStm()
|
  LOOKAHEAD(2)
  AssignmentStm()
|
  LOOKAHEAD(2)
  ArrayAssignmentStm()
}
```

```
void AssignmentStm() :
{}
{ <ID> "=" Expression() ";"   }

void ArrayAssignmentStm() :
{}
{
  <ID> "[" Expression() "]" "="
                Expression() ";"
}
```

Syntactic lookahead

Exp → PrimaryExp *op* *PrimaryExp*		OpExpression
Exp → *PrimaryExp* **[** *Exp* **]**		ArrayLookup
Exp → *PrimaryExp* . **length**		ArrayLength
Exp → *PrimaryExp* . *Id* **(** *ExpList* **)**		MethodCall
Exp → *PrimaryExp*		PrimaryExpression

```
void Expression() :
{}
{
  LOOKAHEAD( PrimaryExpression() <OP> ) OpExpression()
|
 …
  LOOKAHEAD( PrimaryExpression() "[" )  ArrayLookup()
|
  LOOKAHEAD( PrimaryExpression() "." "length" ) ArrayLength()
|
  LOOKAHEAD( PrimaryExpression() "." Identifier() "(" )
  MethodCall()
|
  PrimaryExpression()
}
```

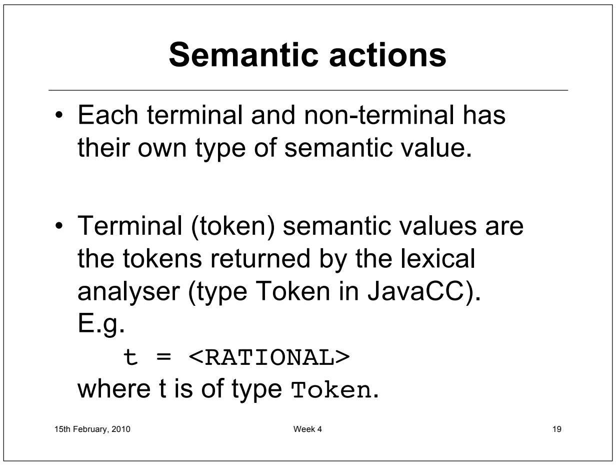
Syntactic lookahead

```
void OpExpression() :
{}
{
  PrimaryExpression() "&&" PrimaryExpression()
}
```

```
void ArrayLength() :
{}
{
  PrimaryExpression() "." "length"
}

void MethodCall() :
{}
{
  PrimaryExpression() "."
Identifier()
  "(" ( ExpressionList() )? ")"
}
```

```
void PrimaryExpression() :
{}
{
  IntegerLiteral()
|
  TrueLiteral()
|
  FalseLiteral()
|
  Identifier()
| …
```

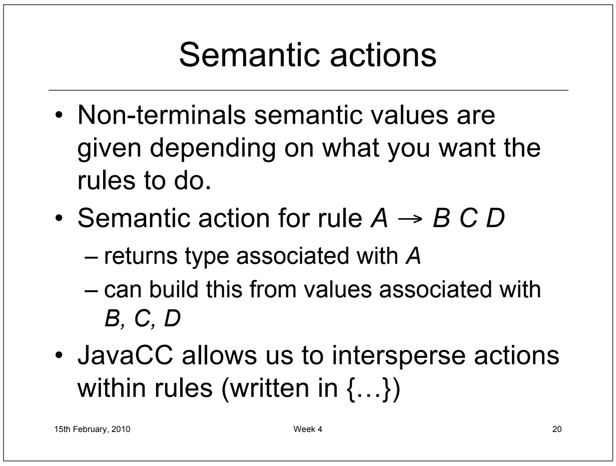


Semantic actions

- Each terminal and non-terminal has their own type of semantic value.
- Terminal (token) semantic values are the tokens returned by the lexical analyser (type Token in JavaCC). E.g.

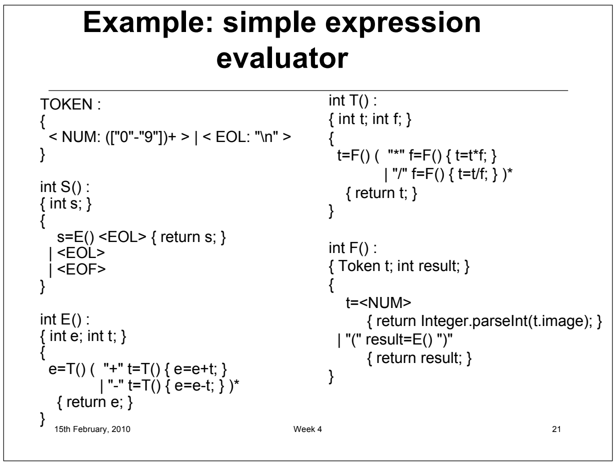
  `t = <RATIONAL>`

  where t is of type `Token`.



Semantic actions

- Non-terminals semantic values are given depending on what you want the rules to do.
- Semantic action for rule *A* → *B C D*
  - returns type associated with *A*
  - can build this from values associated with *B, C, D*
- JavaCC allows us to intersperse actions within rules (written in {…})



Example: simple expression evaluator

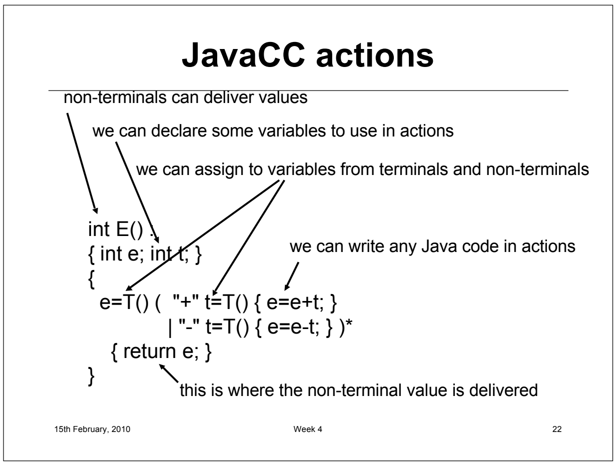
```
TOKEN :
{
  < NUM: (["0"-"9"])+ > | < EOL: "\n" >
}

int S() :
{ int s; }
{
    s=E() <EOL> { return s; }
  | <EOL>
  | <EOF>
}

int E() :
{ int e; int t; }
{
  e=T() (  "+" t=T() { e=e+t; }
            | "-" t=T() { e=e-t; } )*
    { return e; }
}
```

```
int T() :
{ int t; int f; }
{
  t=F() (  "*" f=F() { t=t*f; }
            | "/" f=F() { t=t/f; } )*
    { return t; }
}

int F() :
{ Token t; int result; }
{
    t=<NUM>
        { return Integer.parseInt(t.image); }
  | "(" result=E() ")"
        { return result; }
}
```



JavaCC actions

non-terminals can deliver values

we can declare some variables to use in actions

we can assign to variables from terminals and non-terminals

```
int E() :
{ int e; int t; }
{
  e=T() (  "+" t=T() { e=e+t; }
            | "-" t=T() { e=e-t; } )*
    { return e; }
}
```

we can write any Java code in actions

this is where the non-terminal value is delivered