Procesadores de lenguajes. Práctica 5.

LA HERRAMIENTA JAVACC: ESPECIFICACIÓN SINTÁCTICA

Índice

1.1 Grammar JavaCC

production ::= javacode_production | regular_expr_production

| bnf_production | token_manager_decls

Existen cuatro tipos de producciones en JavaCC:

- javacode_production y bnf_production se utilizan para definir la gramática a partir de la cual se genera el analizador sintáctico.
- regular_expr_production se utiliza para definir los tokens de la gramática - el gestor de tokens se genera a partir de esta información (así como a partir de especificaciones de tokens en línea en la gramática del analizador sintáctico).
- token_manager_decls se utiliza para introducir declaraciones que se insertan en el gestor de tokens generado.

1.2 Javacode_production

javacode_production ::= "JAVACODE"

java_access_modifier java_return_type java_identifier "(" java_parameter_list ")" java_block

- forma de escribir código Java para algunas producciones en lugar de la expansión EBNF habitual.
- Útil para reconocer algo que no está libre de contexto o cuando resulta difícil escribir la gramática.
- Ejemplo el no-terminal skip_to_matching_brace consume tokens de entrada hasta una llave de cierre coincidente.
- Es una caja negra, puede haber problemas en puntos de elección.

1.2 Javacode_production

```
JAVACODE
void skip_to_matching_brace() {
  Token tok;
  int nesting = 1;
  while (true) {
    tok = getToken(1);
    if (tok.kind == LBRACE) nesting++;
    if (tok.kind == RBRACE) {
      nesting--;
      if (nesting == 0) break;
    }
    tok = getNextToken();
  }
}
```

1.2 Javacode_production

```
void NT() : {
}
{
  skip_to_matching_brace()
  |
  some_other_production()
}
```

```
void NT() : {
}
{
  "{" skip_to_matching_brace()
   |
  "(" parameter_list() ")"
}
```

1.2 Javacode_production

- Las producciones JAVACODE en los puntos de elección pueden ir precedidas de LOOKAHEAD sintáctico o semántico.

```
void NT() : {
}
{
  LOOKAHEAD ( {errorOccurred} ) skip_to_matching_brace()
  |
  "(" parameter_list() ")"
}
```

1.3 bnf_production

bnf_production ::= java_access_modifier java_return_type java_identifier "(" java_parameter_list ")" ":“ java_block "{" expansion_choices "}“

- Producción estándar para especificar gramáticas JavaCC.
- Cada producción BNF tiene un lado izquierdo que es una especificación no terminal.
- A continuación, la producción BNF define este no terminal en términos de expansiones BNF en el lado derecho.
- El no terminal se escribe igual que un método Java declarado.
- Cada no terminal se traduce en un método en el analizador sintáctico generado.

1.3 bnf_production

- El nombre del no terminal es el nombre del método, y los parámetros y el valor de retorno declarados son los medios para pasar valores hacia arriba y hacia abajo en el árbol de análisis sintáctico.
- Los no terminales a la derecha de las producciones se escriben cómo llamadas a métodos.
- Hay dos partes a la derecha de una producción BNF:
  - La primera parte es un conjunto de declaraciones y código Java arbitrarios. Se genera al principio del método generado para el no terminal Java.
  - La segunda parte son las expansiones BNF. Las expansiones se representan como una lista de una o más expansiones separadas por '|'.

    expansion_choices ::= expansion ("|" expansion)*

1.4 regular_expr_production

regular_expr_production ::= [lexical_state_list] regexpr_kind ["[" "IGNORE_CASE" "]"] ":“ "{" regexpr_spec ("|" regexpr_spec)* "}“

- Entidades léxicas procesadas por el gestor de tokens generado.
- Primero hay que indicar a los estados léxicos a los que se aplica: “TOKEN” | “SPECIAL_TOKEN” | “SKIP” | “MORE”.
- La opción IGNORE_CASE no distingue entre mayúsculas y minúsculas.
- Por último, una lista de especificaciones de expresión regular.

regexpr_spec ::= regular_expression [java_block] [":" java_identifier]

regular_expression ::= java_string_literal | "<" [["#"] java_identifier ":"] complex_regular_expression_choices ">“ | "<" java_identifier ">“ | "<" "EOF" ">"

1.4 regular_expr_production

```
TOKEN : {
  < FLOATING_POINT_LITERAL:
      (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)? (["f","F","d","D"])?
    | "." (["0"-"9"])+ (<EXPONENT>)? (["f","F","d","D"])?
    | (["0"-"9"])+ <EXPONENT> (["f","F","d","D"])?
    | (["0"-"9"])+ (<EXPONENT>)? ["f","F","d","D"]
  >
  |
  < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
}
```

1.5 token_manager_decls

token_manager_decls ::= "TOKEN_MGR_DECLS" ":" ClassOrInterfaceBody

- Se comienza con la palabra reservada TOKEN_MGR_DECLS seguida de ‘:’ y a continuación un conjunto de declaraciones y sentencias Java.
- Sólo se permite una declaración de gestor de tokens en un archivo de gramática JavaCC.

1.6 local_lookahead

local_lookahead ::= "LOOKAHEAD" "(" [java_integer_literal] [","] [expansion_choices] [","] ["{" java_expression "}"] ")“

- Influye en la forma en la que el analizador sintáctico generado toma decisiones en los distintos puntos de elección de la gramática.
- Comienza con la palabra reservada LOOKAHEAD seguida de restricciones entre paréntesis.
- Hay tres tipos de restricciones lookahead:
  - Un límite lookahead --> literal entero, número máximo tokens para determinar la elección.
  - Una lookahead sintáctica --> opciones de expansión para tomar o no una opción concreta.
  - Una lookahead semántica --> expresión booleana entre llaves, se coge la primera verdadera.

  Si hay más de una, separadas por comas.

1.6.1 Lookahead local

- Valor diferente en aquellos puntos de decisión en los que no sea suficiente con estudiar el siguiente token.
- Ejm de gramática LL(1) con conflicto delante de la opción, cuando el siguiente token es num no se sabe si es el primero o el segundo.

<Llamada> -> id parab (num (coma num)*)? num parce

- Resolución:

```
void llamada() :
 {} {
    <ID> <PARAB> LOOKAHEAD(2) ( <NUM> ( <COMA> <NUM> ) * )? <NUM> <PARCE>
 }
```

1.6 Lookahead sintáctico

```
void JavaDefinition() :
 {} {
     JavaClassDefinition()
   | JavaInterfaceDefinition()
 }
 void JavaClassDefinition() :
 {} {
   (“public” | “abstract” | “final”) * “class”
   JavaClassName()
   JavaClassExtends()
   JavaClassImplements()
   JavaClassBody()
 }

 void JavaInterfaceDefinition() :
 {} {
   (“public” | “abstract” | “final”) * “interface”
   JavaInterfaceName()
   JavaInterfaceExtends()
   JavaInterfaceBody()
 }
```

1.6 Lookahead sintáctico

```
void JavaDefinition() : {} {
    LOOKAHEAD( (“public” | “abstract” | “final”) * “class” ) JavaClassDefinition()  |
JavaInterfaceDefinition()
  }

 void JavaDefinition() : {} {
    LOOKAHEAD( JavaClassDefinition() ) JavaClassDefinition() | JavaInterfaceDefinition()
  }


  void JavaDefinition() :
  {} {
     LOOKAHEAD( 10, (“public” | “abstract” | “final”) * “class” ) JavaClassDefinition()
    | JavaInterfaceDefinition()
  }
```

1.6 Lookahead semántico

```
LOOKAHEAD( { expresión_booleana } )
 void A() :
  {} {
     LOOKAHEAD( { getToken(1).image.startsWith("I") } )
     B()
    | C()
  }
```