Analizador Sintáctico

Garay, Iñaki LU 67387; Touceda, Tomás LU 84024

13 de septiembre de 2011

Analizador Sintáctico
by Garay, Iñaki LU 67387; Touceda, Tomás LU 84024

Índice general

- Consideraciones previas sobre cambios en el analizador lexico, aca habria que comentar los errores que se encontraron en la primera entrega y que cambios se hicieron para que no sucedieran.
- Parsing, aca comentar el tipo de parsing que se implemento, y como era necesario que fuera la gramaica para que pudieramos hacerlo.
- Consideraciones sobre el procesado de la gramatica (lo que esta en el archivo cambios_gramatica, mas/menos alguna que otra correccion/agregado)
- La gramatica final (no se si es mejor poner esto antes del punto de consideraciones... aun que me da igual)
- Una seccion sobre los casos de test, comentando como se muestran los errores y un par de ejemplos.

Capítulo 1

Especificación e instrucciones de uso

El analizador sintáctico esta implementado con Python 2.7.2, por lo cual éste debe estar instalado.
La actual entrega del proyecto consta de los siguiente directorios:

- `./doc/`: contiene la documentacion asociada
- `./tests/lexor/`: contiene los tests de la entrega anterior para la verificación del analizador léxico.
- `./tests/syntaxor/`: contiene los tests de la entrega actual para la verificacion del analizador sintáctico.
- `./src/`: contiene el codigo fuente tanto del analizador léxico como del sintáctico.

Para correr el módulo principal del analizador sintáctico, `syntaxor_main.py`, se debe primero cambiar el directorio actual a `./src/`, el cual contiene a los archivo `*.py`, y ejecutar el comando

```
C:\> python syntaxor_main.py <ARCHIVO DE ENTRADA>
```

A modo de ejemplo, para correr el primero de los tests:

```
C:\> python syntaxor_main.py ../tests/001err.java
```

Si el archivo de código fuente suministrado como entrada no contiene ningún error, se imprimirá un mensaje "La sintaxis de X es correcta."", donde X es el nombre del archivo ingresado. Si por el contrario contiene algún error detectable, el analizador sintáctico generará una excepción e imprimirá un mensaje de error descriptivo.

1.1. Tests automatizados

Para establecer la correctitud del analizador sintáctico, se escribiero una serie de casos de test. Dado que correr los tests cada vez que se realizaba un cambio resultaba engorroso, se desarrollo una manera de correrlos todos automáticamente, indicando si cada test tuvo el resultado esperado o no.
Para correr el test automatizado, cambiar el directorio actual a `./src/` y ejecutar:

```
C:\> python test.py
```

La salida incluirá un mensaje por cada test, indicando si tuvo exito o no, y el mensaje de error emitido por el analizador sintáctico al correr el test.

Capítulo 2

Proceso

La gramática fue procesada de manera tal que fuera LL(1). Para lograr esto primero se eliminó la recursividad a izquierda y luego se factorizó.

El proceso completo consistió en: . Eliminacion de extensiones de la EBNF suministrada por la catedra. La gramatica resultante puede verse en los apendices. . Eliminacion de los aspectos regulares de la gramatica original, los cuales eran resueltos por el analizador léxico. Entre estos estan las reglas para la generacion de identificadores válidos, digitos válidos, operadores, y demas tokens predefinidos y reconocidos. . Eliminacion de reglas recursivas a izquierda, para lograr que la gramatica fuese parseable por un parser descendiente recursivo. . Factorizacion a izquierda para minimizar la cantidad de producciones y simplificar la implementación del parser. . Impementación del parser, implementando una funcion por cada regla de la gramatica. En cada función donde se detectaba una situación anómala, se levanta una excepción con el mensaje correspondiente. . A partir de cada punto de fallo posible, se generó un test diseñado para levantar esa excepción particlar.

Dado que la gramática original no está diseñada para que sea procesada como una LL(1), hubo que realizar algunas excepciones que se trataran en las etapas siguientes:

1. En la gramática original se diferencia entre declaración de constructor y declaración de métodos. Esto sucede porque los constructores no deben llevar un tipo de retorno. Pero dado que el tipo de retorno puede ser un token reconocido como IDENTIFIER, el cual coincide con el tipo de token de nombre del constructor, se resolvió tratar a todos los métodos por igual, y dejar el checkeo de tipo de retorno según el tipo de método para la etapa del análisis semántico.

2. De forma similar a 1., los tipos de los atributos de clase tienen un caso especial: no pueden ser de tipo VOID_TYPE. Esto dificulta la gramática resultante, por lo que se resolvió realizar el checkeo de que los atributos de clase no pueden ser de tipo VOID_TYPE en la etapa de análisis semántico.

3. Otro caso especial ocurre con los constructores, que no pueden poseer el token STATIC en sus modificadores. Como se dijo en 1., se tratarán a los constructores como métodos regulares y el checkeo de que no estén declarados como static se hará en la etapa de análisis semántico.

4. La regla de la gramática original que cuyo lado izquierdo era <expression> contenia una ambiguedad que no es posible resolver desde el punto de vista sintáctico. Esta ambiguedad se debe a la similitud posible entre una sentencia de asignación y una expresión condicional, por lo que se resolvió mover la detección de la asignación más adelante en el arbol sintáctico. Esto da lugar a que sea posible escribir sentencias del estilo: (a + 4) = 2; Este control se realizará en la etapa de análisis semántico.

5. El último cambio significativo a la gramática corresponde a las sentecias de control, específicamente al if. Para resolver ambiguedades en una cadena de ifs anidados con sus respectivos else, la gramática utiliza una serie de reglas que a la hora de convertir a LL(1) resulta imposible. Para saltear esta barrera, se simplificaron las reglas en cuestión, y se asume que cada else está ligado al if más cercano. Esta consideración recién se verá en la etapa de generación de código.

Capítulo 3

Implementación

3.1. Archivos y clases

3.1.1. Archivo `syntaxor_main.py`

Este es el módulo principal del analizador sintáctico, que hace uso del analizador léxico y de la clase Syntaxor para analizar el código MiniJava y establecer su correctitud sintáctica.

3.1.2. Clase Syntaxor

METODOS:

- `check_syntax(self)`: punto de entrada al analizador sintáctico, comienza el proceso de parseo.
- `update_token(self)`: actualiza el token actual que se analiza, solicitandolo al analizador léxico.
- `tok(self, tokentype)`: retorna verdadero si el token actual es del tipo pasado por parametro.

El resto de los metodos de esta clase corresponden cada uno a una de las producciones de la gramatica LL(1).

3.1.3. Clase SyntaxError

La clase `SyntaxError`, definida en el archivo `errors.py`, encapsula el estado asociado a un error. En particular, mantiene la linea y columna donde se detecto el error, y el mensaje de error que se deberá mostrar.

Capítulo 4

Gramatica BNF sin extensiones

Programas

```
<compilation unit> ::= <type declarations>
<compilation unit> ::= LAMBDA
```

Declaraciones

```
<type declarations>               ::= <type declaration>
<type declarations>               ::= <type declarations> <type declaration>
<type declaration>                ::= <class declaration>
<class declaration>               ::= PUBLIC CLASS IDENTIFIER <class body>
<class declaration>               ::= PUBLIC CLASS IDENTIFIER <super> <class body ↩
    >
<super>                           ::= ENTENDS <class type>
<class body>                      ::= BRACE_OPEN BRACE_CLOSE
<class body>                      ::= BRACE_OPEN <class body declarations>  ↩
    BRACE_CLOSE
<class body declarations>         ::= <class body declaration>
<class body declarations>         ::= <class body declarations> <class body  ↩
    declaration>
<class body declaration>          ::= <class member declaration>
<class body declaration>          ::= <constructor declaration>
<class member declaration>        ::= <field declaration>
<class member declaration>        ::= <method declaration>
<constructor declaration>         ::= <constructor modifier> <constructor  ↩
    declarator> <constructor body>
<constructor modifier>            ::= PROTECTED
<constructor modifier>            ::= PUBLIC
<constructor declarator>          ::= <simple type name> PAREN_OPEN PAREN_CLOSE
<constructor declarator>          ::= <simple type name> PAREN_OPEN <formal  ↩
    parameter list> PAREN_CLOSE
<simple type name>                ::= IDENTIFIER
<formal parameter list>           ::= <formal parameter>
<formal parameter list>           ::= <formal parameter list> SEPARATOR(COMMA) < ↩
    formal parameter>
<formal parameter>                ::= <type> <variable declarator id>
<constructor body>                ::= BRACE_OPEN BRACE_CLOSE
<constructor body>                ::= BRACE_OPEN  <block statements>  BRACE_CLOSE
<constructor body>                ::= BRACE_OPEN  <explicit constructor  ↩
    invocation> BRACE_CLOSE
<constructor body>                ::= BRACE_OPEN  <explicit constructor  ↩
    invocation> <block statements> BRACE_CLOSE
<explicit constructor invocation> ::= THIS PAREN_OPEN PAREN_CLOSE SCOLON
<explicit constructor invocation> ::= SUPER PAREN_OPEN PAREN_CLOSE SCOLON
<explicit constructor invocation> ::= THIS PAREN_OPEN <argument list> PAREN_CLOSE ↩
     SCOLON
<explicit constructor invocation> ::= SUPER PAREN_OPEN <argument list>  ↩
    PAREN_CLOSE SCOLON
```

```
<field declaration>               ::= <field modifiers> <type> <variable  ↩
    declarators> SCOLON
<field modifiers>                 ::= <field modifier>
<field modifiers>                 ::= <field modifiers> <field modifier>
<field modifier>                  ::= PUBLIC
<field modifier>                  ::= PROTECTED
<field modifier>                  ::= STATIC
<variable declarators>            ::= <variable declarator>
<variable declarators>            ::= <variable declarators> SEPARATOR(It's a-me  ↩
    a comma) <variable declarator>
<variable declarator>             ::= <variable declarator id>
<variable declarator>             ::= <variable declarator id> ASSIGNMENT < ↩
    variable initializer>
<variable declarator id>          ::= IDENTIFIER
<variable initializer>            ::= <expression>
<method declaration>              ::= <method header> <method body>
<method header>                   ::= <method modifiers> <result type> <method  ↩
    declarator>
<result type>                     ::= <type>
<result type>                     ::= VOID_TYPE
<method modifiers>                ::= <method modifier>
<method modifiers>                ::= <method modifiers> <method modifier>
<method modifier>                 ::= PUBLIC
<method modifier>                 ::= PROTECETED
<method modifier>                 ::= STATIC
<method declarator>               ::= IDENTIFIER PAREN_OPEN PAREN_CLOSE
<method declarator>               ::= IDENTIFIER PAREN_OPEN <formal parameter  ↩
    list> PAREN_CLOSE
<method body>                     ::= <block>
```

Tipos

```
<type>           ::= <primitive type>
<type>           ::= <reference type>
<primitive type> ::= <numeric type>
<primitive type> ::= BOOLEAN_TYPE
<numeric type>   ::= <integral type>
<integral type>  ::= INT
<integral type>  ::= CHAR
<reference type> ::= <class type>
<class type>     ::= <type name>
<type name>      ::= IDENTIFIER
```

Bloques y Sentencias

```
<block>                                  ::= BRACE_OPEN BRACE_CLOSE
<block>                                  ::= BRACE_OPEN <block statements>  ↩
    BRACE_CLOSE
<block statements>                       ::= <block statement>
<block statements>                       ::= <block statements> <block statement ↩
    >
<block statement>                        ::= <local variable declaration  ↩
    statement>
<block statement>                        ::= <statement>
<local variable declaration statement>   ::= <local variable declaration> SCOLON
<local variable declaration>             ::= <type> <variable declarators>
<statement>                              ::= <statement without trailing  ↩
    substatement>
<statement>                              ::= <if then statement>
<statement>                              ::= <if then else statement>
<statement>                              ::= <while statement>
<statement no short if>                  ::= <statement without trailing  ↩
    substatement>
<statement no short if>                  ::= <if then else statement no short if ↩
    >
```

```
<statement no short if>                    ::= <while statement no short if>
<statement without trailing substatement> ::= <block>
<statement without trailing substatement> ::= <empty statement>
<statement without trailing substatement> ::= <expression statement>
<statement without trailing substatement> ::= <return statement>
<empty statement>                          ::= SCOLON
<expression statement>                     ::= <statement expression> SCOLON
<statement expression>                     ::= <assignment>
<statement expression>                     ::= <method invocation>
<statement expression>                     ::= <class instance creation expression ↩
    >
<if then statement>                        ::= IF PAREN_OPEN <expression>  ↩
    PAREN_CLOSE <statement>
<if then else statement>                   ::= IF PAREN_OPEN <expression>  ↩
    PAREN_CLOSE <statement no short if> ELSE <statement>
<if then else statement no short if>       ::= IF PAREN_OPEN <expression>  ↩
    PAREN_CLOSE <statement no short if> ELSE <statement no short if>
<while statement>                          ::= WHILE PAREN_OPEN <expression>  ↩
    PAREN_CLOSE <statement>
<while statement no short if>              ::= WHILE PAREN_OPEN <expression>  ↩
    PAREN_CLOSE <statement no short if>
<return statement>                         ::= RETURN SCOLON
<return statement>                         ::= RETURN <expression> SCOLON
```

Expresiones

```
<expression>                           ::= <assignment expression>
<assignment expression>                ::= <conditional expression>
<assignment expression>                ::= <assignment>
<assignment>                           ::= <left hand side> <assignment operator> < ↩
    assignment expression>
<left hand side>                       ::= <expression name>
<left hand side>                       ::= <field access>
<expression name>                      ::= IDENTIFIER
<assignment operator>                  ::= ASSIGNMENT
<conditional expression>               ::= <conditional or expression>
<conditional or expression>            ::= <conditional and expression>
<conditional or expression>            ::= <conditional or expression>  ↩
    CONDITIONAL_OR <conditional and expression>
<conditional and expression>           ::= <equality expression>
<conditional and expression>           ::= <conditional and expression>  ↩
    CONDITIONAL_AND <equality expression>
<equality expression>                  ::= <relational expression>
<equality expression>                  ::= <equality expression> EQUALS <relational ↩
     expression>
<equality expression>                  ::= <equality expression> NOT_EQUALS < ↩
    relational expression>
<relational expression>                ::= <additive expression>
<relational expression>                ::= <relational expression> LT <additive  ↩
    expression>
<relational expression>                ::= <relational expression> GT <additive  ↩
    expression>
<relational expression>                ::= <relational expression> LT_EQ <additive  ↩
    expression>
<relational expression>                ::= <relational expression> GT_EQ <additive  ↩
    expression>
<additive expression>                  ::= <multiplicative expression>
<additive expression>                  ::= <additive expression> ADD < ↩
    multiplicative expression>
<additive expression>                  ::= <additive expression> SUB < ↩
    multiplicative expression>
<multiplicative expression>            ::= <unary expression>
<multiplicative expression>            ::= <multiplicative expression> MUL <unary  ↩
    expression>
```

```
<multiplicative expression>          ::= <multiplicative expression> DIV <unary  ↩
    expression>
<multiplicative expression>          ::= <multiplicative expression> MOD <unary  ↩
    expression>
<unary expression>                   ::= ADD <unary expression>
<unary expression>                   ::= SUB <unary expression>
<unary expression>                   ::= <unary expression not plus minus>
<unary expression not plus minus>    ::= <postfix expression>
<unary expression not plus minus>    ::= NOT <unary expression>
<postfix expression>                 ::= <primary>
<postfix expression>                 ::= <expression name>
<primary>                            ::= <literal>
<primary>                            ::= THIS
<primary>                            ::= PAREN_OPEN <expression> PAREN_CLOSE
<primary>                            ::= <class instance creation expression>
<primary>                            ::= <field access>
<primary>                            ::= <method invocation>
<class instance creation expression> ::= NEW <class type> PAREN_OPEN PAREN_CLOSE
<class instance creation expression> ::= NEW <class type> PAREN_OPEN <argument  ↩
    list> PAREN_CLOSE
<argument list>                      ::= <expression>
<argument list>                      ::= <argument list> SEPARATOR(comma) < ↩
    expression>
<field access>                       ::= <primary> ACCESSOR IDENTIFIER
<field access>                       ::= SUPER ACCESSOR IDENTIFIER
<method invocation>                  ::= <method name> PAREN_OPEN PAREN_CLOSE
<method invocation>                  ::= <method name> PAREN_OPEN <argument list> ↩
     PAREN_CLOSE
<method invocation>                  ::= <primary> ACCESSOR IDENTIFIER PAREN_OPEN ↩
     PAREN_CLOSE
<method invocation>                  ::= <primary> ACCESSOR IDENTIFIER PAREN_OPEN ↩
     <argument list> PAREN_CLOSE
<method invocation>                  ::= SUPER ACCESOR IDENTIFIER PAREN_OPEN  ↩
    PAREN_CLOSE
<method invocation>                  ::= SUPER ACCESOR IDENTIFIER PAREN_OPEN < ↩
    argument list> PAREN_CLOSE
<method name>                        ::= IDENTIFIER
<literal>                            ::= INT_LITERAL
<literal>                            ::= <boolean literal>
<literal>                            ::= CHAR_LIT
<literal>                            ::= STRING_LIT
<literal>                            ::= NULL
<boolean literal                     ::= TRUE
<boolean literal                     ::= FALSE
```

Capítulo 5

Gramatica LL(1) resultante

Programas

```
<compilation unit> ::= <type declarations>
```

Declaraciones

```
<type declarations>             ::= <class declaration> <type declarations>
<type declarations>             ::= LAMBDA
<class declaration>             ::= PUBLIC CLASS IDENTIFIER <rest class  ↩
    declaration>
<rest class declaration>        ::= <class body>
<rest class declaration>        ::= ENTENDS IDENTIFIER <class body>
<class body>                    ::= BRACE_OPEN <rest class body>
<rest class body>               ::= BRACE_CLOSE
<rest class body>               ::= <class body declarations> BRACE_CLOSE
<class body declarations>       ::= <class body declaration> <rest class body  ↩
    declarations>
<rest class body declarations> ::= LAMBDA
<rest class body declarations> ::= <class body declarations>
<class body declaration>        ::= <field modifiers> <rest class body declaration ↩
    >
<rest class body declaration>  ::= <type noident void> <declarators>
<rest class body declaration>  ::= IDENTIFIER <rest2 class body declaration>
<rest2 class body declaration> ::= <constructor declarator> <constructor body>
<rest2 class body declaration> ::= <declarators>
<constructor declarator>        ::= PAREN_OPEN <rest constructor declarator>
<rest constructor declarator>  ::= PAREN_CLOSE
<rest constructor declarator>  ::= <formal parameter list> PAREN_CLOSE
<formal parameter list>         ::= <formal parameter> <rest formal parameter list ↩
    >
<rest formal parameter list>   ::= COMMA <formal parameter list>
<rest formal parameter list>   ::= LAMBDA
<formal parameter>              ::= <type> IDENTIFIER
<constructor body>              ::= BRACE_OPEN <rest constructor body>
<rest constructor body>         ::= BRACE_CLOSE
<rest constructor body>         ::= <block statements> BRACE_CLOSE
<rest2 constructor body>        ::= BRACE_CLOSE
<rest2 constructor body>        ::= <block statements> BRACE_CLOSE
<field modifiers>               ::= <field modifier> <rest field modifiers>
<rest field modifiers>          ::= LAMBDA
<rest field modifiers>          ::= <field modifiers>
<field modifier>                ::= PUBLIC
<field modifier>                ::= PROTECTED
<field modifier>                ::= STATIC
<declarators>                   ::= IDENTIFIER <rest declarators>
<rest declarators>              ::= COMMA <rest declarators>
<rest declarators>              ::= ASSIGNMENT <expression> SCOLON
<rest declarators>              ::= PAREN_OPEN <rest method declarator> <method  ↩
    body>
```

```
<rest declarators>              ::= SCOLON
<rest method declarator>        ::= PAREN_CLOSE
<rest method declarator>        ::= <formal parameter list> PAREN_CLOSE
<method body>                   ::= <block>
```

Tipos

```
<type>              ::=                         <primitive type>
<type>              ::=                         IDENTIFIER
<type>              ::=                         VOID_TYPE
<type noident void> ::=                         <primitive type>
<type noident void> ::=                         VOID_TYPE
<primitive type>    ::=                         <numeric type>
<primitive type>    ::=                         <boolean type>
<numeric type>      ::=                         <integral type>
<integral type>     ::= INT_TYPE
<integral type>     ::= CHAR_TYPE
<boolean type>      ::= BOOLEAN_TYPE
```

Bloques y Sentencias

```
<block>                                     ::= BRACE_OPEN <rest block>
<rest block>                                ::= BRACE_CLOSE
<rest block>                                ::= <block statements> BRACE_CLOSE
<block statements>                          ::= <block statement> <rest block  ↩
    statements>
<rest block statements>                     ::= LAMBDA
<rest block statements>                     ::= <block statements>
<block statement>                           ::= <primitive type> <local variable  ↩
    declaration statement>
<block statement>                           ::= <if start statement>
<block statement>                           ::= <while statement>
<block statement>                           ::= <block>
<block statement>                           ::= <empty statement>
<block statement>                           ::= <return statement>
<block statement>                           ::= <primary> <rest method invocation>
<local variable declaration statement>      ::= <local variable declaration> SCOLON
<local variable declaration>                ::= IDENTIFIER <variable declarators>
<variable declarators>                      ::= <variable declarator> <rest  ↩
    variable declarators>
<rest variable declarators>                 ::= LAMBDA
<rest variable declarators>                 ::= COMMA IDENTIFIER <variable  ↩
    declarators>
<rest variable declarators>                 ::= COMMA THIS <variable declarators>
<rest variable declarators>                 ::= COMMA SUPER <variable declarators>
<variable declarator>                       ::= <rest variable declarator>
<rest variable declarator>                  ::= LAMBDA
<rest variable declarator>                  ::= ASSIGNMENT <expression>
<statement>                                 ::= <statement without trailing  ↩
    substatement>
<statement>                                 ::= <if start statement>
<statement>                                 ::= <while statement>
<statement without trailing substatement>   ::= <block>
<statement without trailing substatement>   ::= <empty statement>
<statement without trailing substatement>   ::= <expression statement>
<statement without trailing substatement>   ::= <return statement>
<empty statement>                           ::= SCOLON
<expression statement>                      ::= <statement expression> SCOLON
<statement expression>                      ::= <method invocation>
<if start statement>                        ::= IF PAREN_OPEN <expression>  ↩
    PAREN_CLOSE <statement> <rest if start statement>
<rest if start statement>                   ::= LAMBDA
<rest if start statement>                   ::= ELSE <statement>
<while statement>                           ::= WHILE PAREN_OPEN <expression>  ↩
    PAREN_CLOSE <statement>
```

```
<return statement>                          ::= RETURN <rest return statement>
<rest return statement>                     ::= SCOLON
<rest return statement>                     ::= <expression> SCOLON
```

Expresiones

```
<expression>                                ::= <assignment expression>
<assignment expression>                     ::= <conditional expression>
<conditional expression>                    ::= <conditional or expression> <rest  ↩
    conditional expression>
<rest conditional expression>               ::= LAMBDA
<rest conditional expression>               ::= ASSIGNMENT <conditional expression>
<conditional or expression>                 ::= <conditional and expression> <rest  ↩
    conditional or expression>
<rest conditional or expression>            ::= CONDITIONAL_OR <conditional or  ↩
    expression>
<rest conditional or expression>            ::= LAMBDA
<conditional and expression>                ::= <equality expression> <rest  ↩
    conditional and expression>
<rest conditional and expression>           ::= LAMBDA
<rest conditional and expression>           ::= CONDITIONAL_AND <conditional and  ↩
    expression>
<equality expression>                       ::= <relational expression> <rest  ↩
    equality expression>
<rest equality expression>                  ::= EQUALS <equality expression>
<rest equality expression>                  ::= NOT_EQUALS <equality expression>
<rest equality expression>                  ::= LAMBDA
<relational expression>                     ::= <additive expression> <rest  ↩
    relational expression>
<rest relational expression>                ::= LT <relational expression>
<rest relational expression>                ::= GT <relational expression>
<rest relational expression>                ::= LT_EQ <relational expression>
<rest relational expression>                ::= GT_EQ <relational expression>
<rest relational expression>                ::= LAMBDA
<additive expression>                       ::= <multiplicative expression> <rest  ↩
    additive expression>
<rest additive expression>                  ::= ADD <additive expression>
<rest additive expression>                  ::= SUB <additive expression>
<rest additive expression>                  ::= LAMBDA
<multiplicative expression>                 ::= <unary expression> <rest  ↩
    multiplicative expression>
<rest multiplicative expression>            ::= MUL <multiplicative expression>
<rest multiplicative expression>            ::= DIV <multiplicative expression>
<rest multiplicative expression>            ::= MOD <multiplicative expression>
<rest multiplicative expression>            ::= LAMBDA
<unary expression>                          ::= ADD <unary expression>
<unary expression>                          ::= SUB <unary expression>
<unary expression>                          ::= <unary expression not plus minus>
<unary expression not plus minus>           ::= <postfix expression>
<unary expression not plus minus>           ::= NOT <unary expression>
<postfix expression>                        ::= <primary>
<primary>                                   ::= <literal> <rest primary>
<primary>                                   ::= THIS <rest primary>
<primary>                                   ::= PAREN_OPEN <expression> PAREN_CLOSE ↩
     <rest primary>
<primary>                                   ::= <class instance creation expression ↩
    > <rest primary>
<primary>                                   ::= SUPER ACCESSOR IDENTIFIER <rest  ↩
    primary>
<primary>                                   ::= <method invocation> <rest primary>
<rest primary>                              ::= ACCESSOR IDENTIFIER <rest2 primary>
<rest primary>                              ::= LAMBDA
<rest2 primary>                             ::= PAREN_OPEN <rest2 method invocation ↩
    > <rest primary>
<rest2 primary>                             ::= LAMBDA
```

```
<class instance creation expression>      ::= NEW IDENTIFIER PAREN_OPEN <rest  ↩
    class instance creation expression>
<rest class instance creation expression> ::= PAREN_CLOSE
<rest class instance creation expression> ::= <argument list> PAREN_CLOSE
<argument list>                           ::= <expression> <rest argument list>
<rest argument list>                      ::= LAMBDA
<rest argument list>                      ::= COMMA <argument list>
<method invocation>                       ::= IDENTIFIER <rest primary> <rest  ↩
    method invocation>
<method invocation>                       ::= <literal> <rest primary> ACCESSOR  ↩
    IDENTIFIER PAREN_OPEN <rest2 method invocation> <rest method invocation>
<method invocation>                       ::= THIS <rest primary> ACCESSOR  ↩
    IDENTIFIER PAREN_OPEN <rest2 method invocation> <rest method invocation>
<method invocation>                       ::= PAREN_OPEN <expression> PAREN_CLOSE ↩
     <rest primary> ACCESSOR IDENTIFIER PAREN_OPEN <rest2 method invocation> < ↩
    rest method invocation>
<method invocation>                       ::= <class instance creation expression ↩
    > <rest primary> ACCESSOR IDENTIFIER PAREN_OPEN <rest2 method invocation> < ↩
    rest method invocation>
<method invocation>                       ::= SUPER <rest primary> <rest super>
<rest super>                              ::= IDENTIFIER PAREN_OPEN <rest2 method ↩
     invocation> <rest method invocation>
<rest super>                              ::= PAREN_OPEN <rest2 method invocation ↩
    > <rest method invocation>
<rest method invocation>                  ::= ACCESSOR IDENTIFIER PAREN_OPEN < ↩
    rest2 method invocation> <rest method invocation>
<rest method invocation>                  ::= <variable declarators>
<rest method invocation>                  ::= <rest variable declarators>
<rest method invocation>                  ::= PAREN_OPEN <rest2 method invocation ↩
    > <rest method invocation>
<rest method invocation>                  ::= LAMBDA
<rest2 method invocation>                 ::= PAREN_CLOSE
<rest2 method invocation>                 ::= <argument list> PAREN_CLOSE
<boolean literal>                         ::= TRUE
<boolean literal>                         ::= FALSE
```