ST0244
Lenguajes de Programación

Departamento de Informática y Sistemas
Universidad EAFIT

1

Tópicos

- Gramática del lenguaje (parcial)
- Scanner
- Parser
- ¿Qué hay que hacer?
- Qué hay que entregar

2

Gramática del lenguaje

- Gramática en EBNF (Extended Backus-Naur Form)
- <noTerminal>
- terminal
- ::= (definido como)
- | (alternativa)
- {} (cero o más ocurrencias)
- [] (opcional)
- () (opciones a seleccionar)

<program> ::= program <funDefinitionList>
endProgram
<funDefinitionList> ::= <funDefinition> { <funDefinition> }
<funDefinition> ::= def <variable> <lparen> [<varDefList>]
<rparen> [<varDefList>] [<statementList>] enddef
<varDefList> ::= <varDef> { <varDef> }
<varDef> ::= int <variable>
<statementList> ::= <statement> { <statement>}
<statement> ::= read <variable> | print <variable> |
call <variable> <lparen> [<argumentList>] <rparen>
<argumentList> ::= <argumentDef> {<argumentDef>}
<argumentDef> ::= <variable>
<lparen> ::= (
<rparen> ::)

3

Gramática del lenguaje

Mirar el programa fuente y generarlo a partir de la gramática:

```
program

  def main ( int a int b )
    int d
    read d
    call sumar ( a b )
    print a
  enddef

  def sumar ( int x int y )
    int z
  enddef

endprogram
```

<program> ::= program <funDefinitionList>
endProgram
<funDefinitionList> ::= <funDefinition> { <funDefinition> }
<funDefinition> ::= def <variable> <lparen> [<varDefList>]
<rparen> [<varDefList>] [<statementList>] enddef
<varDefList> ::= <varDef> { <varDef> }
<varDef> ::= int <variable>
<statementList> ::= <statement> { <statement>}
<statement> ::= read <variable> | print <variable> |
call <variable> <lparen> [<argumentList>] <rparen>
<argumentList> ::= <argumentDef> {<argumentDef>}
<argumentDef> ::= <variable>
<lparen> ::= (
<rparen> ::)

4

Gramática del lenguaje

Mirar, por ejemplo, una lista de definiciones de variables.
Derivación por izquierda:

<varDefList> ::= <varDef> { <varDef> }
<varDef> ::= int <variable>

<varDefList> -> <varDef><varDefList> -> int i <varDefList> -> int i <varDef><varDefList> -> int i int j <varDefList> -> int i int j

De manera similar, se generan listas de instrucciones, listas de parámetros, etc…

Recordar que "épsilon" es símbolo vacío, no se consumen símbolos de la entrada

Scanner

- Es un scanner muy sencillo, escrito en Java.
- Cada "token" debe estar separado por blancos. Por esto los programas se ven así:

```
program

  def main ( int a int b )
    int d
    read d
    call sumar ( a b )
    print a
  enddef

  def sumar ( int x int y )
    int z
  enddef

endprogram
```

Scanner (clase Lexer)

- Notar las constantes, que se usan desde el Parser. En el proyecto se van agregar más.

```
public static final int PROGRAM = 1;
public static final int ENDPROGRAM = 2;
public static final int DEF = 3;
public static final int ENDDEF = 4;
public static final int IF = 5;
public static final int ELSE = 6;
public static final int ENDIF = 7;
public static final int VARIABLE = 8;
public static final int CONSTANT = 9;
public static final int EQUALS = 10;
public static final int ASSIGN = 11;
public static final int LPAREN = 12;
public static final int RPAREN = 13;
public static final int INT = 14;
public static final int READ = 15;
public static final int PRINT = 16;
public static final int CALL = 17;
public static final int INVALIDTOKEN = 98;
public static final int EOF = 99;
```

7

Scanner (clase Lexer)

- Notar la tabla, donde se buscan las palabras claves

```
//Initialize the keyword table
keywordsTable = new ArrayList<Token>();
keywordsTable.add(new Token(PROGRAM, "program", 0));
keywordsTable.add(new Token(ENDPROGRAM, "endprogram", 0));
keywordsTable.add(new Token(DEF, "def", 0));
keywordsTable.add(new Token(ENDDEF, "enddef", 0));
keywordsTable.add(new Token(IF, "if", 0));
keywordsTable.add(new Token(ELSE, "else", 0));
keywordsTable.add(new Token(ENDIF, "endif", 0));
keywordsTable.add(new Token(EQUALS, "==", 0));
keywordsTable.add(new Token(LPAREN, "(", 0));
keywordsTable.add(new Token(RPAREN, ")", 0));
keywordsTable.add(new Token(INT, "int", 0));
keywordsTable.add(new Token(READ, "read", 0));
keywordsTable.add(new Token(PRINT, "print", 0));
keywordsTable.add(new Token(CALL, "call", 0));
keywordsTable.add(new Token(EOF, "EOF", 0));
```

8

Scanner

- El corazón del Scanner es este método
- Retorna un "token", que debe estar separado por espacios en el programa fuente

```
public Lexer (String fileName) throws FileNotFoundException {

    //Define an scanner for the source file
    fileScanner = new Scanner(new File(fileName));

    //Initialize the linecount
    lineCount=0;

    //Initialize the keyword table
    keywordsTable = new ArrayList<Token>();
    keywordsTable.add(new Token(PROGRAM, "program", 0));
    keywordsTable.add(new Token(ENDPROGRAM, "endprogram", 0));
    keywordsTable.add(new Token(DEF, "def", 0));
    keywordsTable.add(new Token(ENDDEF, "enddef", 0));
    keywordsTable.add(new Token(IF, "if", 0));
    keywordsTable.add(new Token(ELSE, "else", 0));
    keywordsTable.add(new Token(ENDIF, "endif", 0));
    keywordsTable.add(new Token(EQUALS, "==", 0));
    keywordsTable.add(new Token(LPAREN, "(", 0));
    keywordsTable.add(new Token(RPAREN, ")", 0));
    keywordsTable.add(new Token(INT, "int", 0));
    keywordsTable.add(new Token(READ, "read", 0));
    keywordsTable.add(new Token(PRINT, "print", 0));
    keywordsTable.add(new Token(CALL, "call", 0));
    keywordsTable.add(new Token(EOF, "EOF", 0));


    /** extract each token from the source file **/
    String tokenText;
    int tokenCode;
    //create the tokenList
    tokenList = new ArrayList<Token>();
    //go through the source file
    do{
        //Obtain the next token
        tokenText = nextText();
        //Classify each token
        tokenCode = getTokenCode(tokenText);
        //print token's data
        System.out.println(tokenText + " code: " + tokenCode + " line:" + lineCount);
        //add token to the tokenList
        tokenList.add(new Token(tokenCode, tokenText, lineCount));
    }
    while(tokenText.compareTo("EOF")!=0);
```


9

Parser

Notar el atributo "toke
donde se tiene el siguiente token a identificar

Notar el constructor: se crea el Scanner y se lee el primer "token"

Notar el método recognize(), donde se consume un token del programa fuente

```
public class Parser{
    //token: (Token) used to process each token in the source code
    private Token token;
    //lexer: (Lexer) used to obtain each token in the source code
    private Lexer lexer;

    /**
        Constructor:
            Creates the lexer to obtain each token in the source file
            Points the iterator to the first token
            Starts parsing at the start symbol according the the grammar
    **/
    public Parser(String fileName){
        try{
            //Creates the lexer
            lexer = new Lexer(fileName);
            //Points the iterator to the first token
            token = lexer.nextToken();
            //Starts the parser
            program();
        } catch (FileNotFoundException ex){
            System.out.println("File not found!");
            System.exit(1);
        }
    }
    /**
        Function recognize: verifies if the current token corresponds to the expected token
        according with the grammar and move the tokenIterator to the next one.
        @param expected: (int) expected token code
    **/
    private void recognize(int expected){
        if (token.code == expected)
            //the token is the expected one, move the tokenIterator
            token = lexer.nextToken();
        else {
            //The current token is not expected, syntax error!
            System.out.println("Syntax error in line " + token.line);
            System.out.println("Expected: "  + expected + " found " + token.code);
            //STOP!
            System.exit(2);
        }
    }
```


10

Parser

- Notar cómo se traducen estas producciones de la gramática al código correspondiente:

```
/**
    Function program: verifies the production
        <program>::= program <funDefinitionList> endProgram
**/
public void program(){
    //checks for "program"
    recognize(Lexer.PROGRAM);
    //checks for <funDefinitionList>
    funDefinitionList();
    //checks for "endProgram"
    recognize(Lexer.ENDPROGRAM);
    //if EOf is found, no error is found!
    if (token.code == Lexer.EOF){
        System.out.println("No errors found!");
    }
}
```

```
/**
    Function funDefinitionList: verifies the production
    <funDefinitionList>::=<funDefinition> { <funDefinition> }
**/
public void funDefinitionList()
{
    //checks for <funDefinition>
    funDefinition();
    //verifies if there are more <funDefinition>
    while(lexer.getCurrentToken().code == Lexer.DEF){
        funDefinition();
    }
}

/**
    Function funDefinition: verifies the production
    <funDefinition>::= def <variable> <lparen> [<varDefList>] <rparen>
                                [<varDefList>]
                                [<statementList>]
                            enddef
**/
public void funDefinition(){
    //checks for "def"
    recognize(Lexer.DEF);
    //checks for <variable>
    recognizeVariable();
    //checks for "("
    recognize(Lexer.LPAREN);
    //verifies if there are parameters
    if (lexer.getCurrentToken().code != Lexer.RPAREN)
        varDefList();
    //checks for ")"
    recognize(Lexer.RPAREN);
    //ver is there are variables definition
    if (lexer.getCurrentToken().code == Lexer.INT)
        varDefList();
    //checks for <statementList>
    statementList();
    //checks for "enddef"
    recognize(Lexer.ENDDEF);
}
```


¿Qué hay que hacer?

- Hay que agregar la siguiente funcionalidad:
  - Expresiones aritméticas con variables o constantes numéricas y con signos ‘+’ y ‘*’ (no hay operadores ‘-’ ni ‘/’)
  - Condiciones: solo preguntar por igual y por diferente (no hay operadores && ni ||).
  - Asignaciones (variable = expresión aritmética)
  - Decisiones (if_then_else)
  - Ciclos (ciclo while)
  - Definir variables y constantes
  - Cada grupo define la sintaxis que quiera

¿Qué hay que hacer?

- Para las expresiones aritméticas usar las siguientes producciones:

```
<expr> ::= <term> { + <term>}
<term> ::= <factor>{* <factor>}
<factor> ::= ( <expr> ) |
    variable | constant
```

- Notar que esta gramática maneja bien la precedencia. ¿Cómo? Explicar

¿Qué hay que hacer?

Se sugiere el siguiente procedimiento:

Para cada nueva funcionalidad:

1. Inventarse la sintaxis (excepto para las expresiones aritméticas, que ya está definida)
2. Definir la(s) producción(es) correspondiente(s)
3. Agregar la(s) palabra(s) clave en el Scanner (si hace falta)
4. Agregar el código en el Parser
5. Probar con un programa fuente correcto y uno incorrecto

Repetir los 5 pasos para cada funcionalidad nueva

A tener en cuenta

- Las instrucciones se deben poder anidar. Esto es: un ciclo dentro de otro, o un if dentro de un ciclo, etc.
- La sintaxis debe seguir más o menos el estilo de lo definid hasta ahora

15

Qué se debe entregar: 3 cosas
Fecha de entrega: 14 septiembre

1. Documento. Para cada funcionalidad adicional, 3 columnas así:

Producciones de la gramática	Código del Parser	Ejemplo correcto Ejemplo con error

2. Para cada funcionalidad adicional, un clip de video de unos segundos que muestre el Parser funcionando con un ejemplo correcto y con un ejemplo incorrecto

16

Qué se debe entregar: 3 cosas

- 3a. Se debe escribir un programa, en su lenguaje, que sea equivalente al siguiente código en Java.
- 3b. Se debe crear un clip de video mostrando el programa en su lenguaje y mostrando que el Parser no encuentra errores de sintaxis

```
Scanner in = new Scanner(System.in);
int i;
int acum;
int j;
i = in.nextInt();
acum = 0;
j = 0;
if(i == 10) {
    while(j != i) {
        acum = acum + j * (i + 1);
        j = j + 1;
    }
} else {
    acum = 1;
}
System.out.println(acum);
```