Gramática Sintáctica

(Lenguaje Micro)

<programa> → **inicio** <listaSentencias> **fin**

<listaSentencias> → <sentencia> {<sentencia>}

<sentencia> → <identificador> **:=** <expresión> **;** |

leer (<listaIdentificadores>) **;** |

escribir (<listaExpresiones>) **;**

<listaIdentificadores> → <identificador> {, <identificador>}

<listaExpresiones> → <expresión> {, <expresión>}

<expresión> → <primaria> {<operadorAditivo> <primaria>}

<primaria> → <identificador> | <constante> | **(** <expresión> **)**

Parser

- Hay varios tipos de parsers. Para este ejemplo usaremos el “Análisis Sintáctico Descendente Recursivo” (ASDR)
- Un ASDR es un parser del tipo top-down, es decir, parte del axioma y va armando el Árbol de Análisis Sintáctico (AAS) consistente con una derivación a izquierda.
- Otros parsers son bottom-up es decir, a partir de los terminales (hojas del árbol) van reduciendo hacia el axioma.
- El AAS queda de modo tal que en las hojas están los terminales y en los nodos interiores los no terminales.
- Cada nodo tiene como hijos la secuencia de terminales y no terminales que representa una posible producción de ese no terminal.
- El AAS no siempre es una estructura de datos. En el caso del ASDR el árbol es virtualmente armando por los llamados recursivos entre funciones

ASDR

- La idea es que por cada No Terminal haya un Procedimiento de Análisis Sintáctico (PAS) que lo implemente, usando el mismo nombre.
- Si el No terminal tiene una única producción la implementación es muy simple
  - Por cada No Terminal llamamos a las PAS correspondiente (y acá está la parte recursiva)
  - Por cada terminal llamamos una función Match(t) que compruebe que el terminal que provee el scanner coincida con el terminal que la producción prevee.
- Si el No Terminal tiene más de una producción se agregará código adicional para manejar la situación (veremos con el ejemplo del lenguaje Micro)

ASDR

- Primero agregaremos un nuevo axioma que incluya en consideración el fdt.
- También reemplazamos los Terminales por las enumeraciones que los representan. Esto NO es necesario, es solamente una cuestión didáctica, para hacerla más coincidente con los PAS que se verán luego
- IMPORTANTE: Match pertenece al scanner y es la **única** función que hace avanzar al mismo
  - Compara el token a coincidir con el próximo símbolo a entregar por el scanner.
  - Si coinciden se sigue el análisis sin problemas
  - Si no coinciden implica un **error sintáctico**
  - En ambos casos habilita al scanner a avanzar leyendo un nuevo token

Gramática Sintáctica modificada

<objetivo> → <programa> FDT

<programa> → INICIO <listaSentencias> FIN

<listaSentencias> → <sentencia> {<sentencia>}

<sentencia> → ID ASIGNACION <expresión> PUNTOYCOMA |

LEER PARENIZQUIERDO <listaIdentificadores> PARENDERECHO PUNTOYCOMA |

ESCRIBIR PARENIZQUIERDO <listaExpresiones> PARENDERECHO PUNTOYCOMA

<listaIdentificadores> → ID {COMA ID}

<listaExpresiones> → <expresión> {COMA <expresión>}

<expresión> → <primaria> {<operadorAditivo> <primaria>}

<primaria> → ID |

CONSTANTE |

PARENIZQUIERDO <expresión> PARENDERECHO

<operadorAditivo> → *uno de* SUMA RESTA

PAS de producciones simples

Por cada No Terminal llamamos al PAS correspondiente y por cada Terminal llamamos Match(Terminal)

```
void Objetivo (void) {
    // <objetivo> -> <programa> FDT
    Programa();
    Match(FDT);
}

void Programa (void) {
    // <programa> → INICIO <listaSentencias> FIN
    Match(INICIO);
    ListaSentencias();
    Match(FIN);
}
```

Producciones con repeticiones

- En los casos como <listaSentencias> o como <listaIdentificadores> tenemos un comienzo simple, pero luego se puede repetir opcionalmente una cantidad desconocida de veces
- Para saber si debemos repetir o no usaremos la función ProximoToken() que devuelve cuál es el token siguiente (sin avanzar el scanner, que **solo avanza** con la función **Match**)
- El ejemplo que estamos viendo tiene una particularidad en su GIC y es que es de tipo LL(1) lo que permite que viendo solamente el próximo token se pueda tomar la decisión

PAS ListaSentencias

```
void ListaSentencias (void) {
// <listaSentencias> -> <sentencia> {<sentencia>}
    Sentencia(); // la primera de la lista
    while (1) {  // un ciclo indefinido
        switch (ProximoToken()) {
        case ID:
        case LEER:
        case ESCRIBIR: // detectó token correcto
            Sentencia(); // procesa la secuencia
                         // opcional
            break;
        default:
            return;
        } // fin switch
    }
}
```

PAS producciones con opciones

- Si el No Terminal Tiene varias producciones hay que verificar si lo que sigue coincide con una de ellas (y determinar cuál de todas).
- Dado que la gramática que usamos es LL(1) podemos saberlo leyendo un token por adelantado con ProximoToken()
- Si el token leído no es ninguno de los esperados, entonces se produce un error sintáctico

PAS Sentencia

```
void Sentencia(void) {
    TOKEN tok = ProximoToken();
    switch (tok) {
    case ID: // <sentencia> -> ID := <expresion>;
        Match(ID); Match(ASIGNACION);
        Expresión(); Match(PUNTOYCOMA);
        break;
    case LEER: // <sentencia> -> LEER ( <listaIdentificadores> );
        Match(LEER); Match(PARENIZQUIERDO);
        ListaIdentificadores();
        Match(PARENDERECHO); Match(PUNTOYCOMA);
        break;
    case ESCRIBIR: // <sentencia> -> ESCRIBIR (<listaExpresiones>);
        Match(ESCRIBIR); Match(PARENIZQUIERDO);
        ListaExpresiones();
        Match(PARENDERECHO); Match(PUNTOYCOMA);
        break;
    default:
        ErrorSintactico(tok); break;
    }
}
```

Más ejemplos de PAS

Otro ejemplo de PAS con repeticiones

```
void listaExpresiones (void) {
// <listaExpresiones> → <expresión> {COMA <expresión>}
    Expresion(); // la primera de la lista de expresiones
    while (ProximoToken() == COMA) {// El resto de las
                                    // opcionales
        Match(COMA); Expresion();
    }
}
```

Más ejemplos de PAS

Otro ejemplo de PAS con opciones

```
void Primaria(void) {
// <primaria> → ID | CONSTANTE |
//                PARENIZQUIERDO <expresión> PARENDERECHO
    TOKEN tok = ProximoToken();
    switch (tok) {
    case ID:
        Match(ID); break;
    case CONSTANTE:
        Match(CONSTANTE); break;
    case PARENIZQUIERDO:
        Match(PARENIZQUIERDO); Expresion();
        Match(PARENDERECHO); break;
    default:
        ErrorSintactico(tok); break;
    }
}
```

Análisis Semántico

- El analizador semántico, o generador de código intermedio, se encarga de:
  - Hacer los controles que son propios de una gramática sensible al contexto:
    - Comprobar si las variables están definidas
    - Comprobar el tipo de las variables
    - Comprobar tipo y cantidad de parámetros de una función
    - etc
  - Ir generando código de una máquina virtual (MV) que el compilador usa como intermedio antes de generar código para una arquitectura de procesador específica

Análisis Semántico

- El parser es quien va llamando al analizador semántico en momentos determinados, de modo similar a como hace con el escáner
- Para ello primero se modifica la gramática y se agregan **símbolos de acción**. Para cada símbolo de acción se programará su correspondiente **rutina semántica**.
  - La gramática con los símbolos de acción agregados se conoce como gramática con anotaciones
- Los símbolos se agregan en el punto en el cuál debe llamarse a la rutina semántica. Los indicamos con un # adelante
- Para que las rutinas semánticas puedan trabajar necesitan datos.
  - Desde el punto de vista formal se conocen como atributos
  - Desde el punto de vista práctico se conocen como registros semánticos

Rutinas semánticas

- Donde ubicar las rutinas semánticas
  - Donde haya que hacer chequeos de semántica estática (en nuestro caso solo declarar las variables cuando aparecen por primera vez)
  - Donde haya que generar código
  - Donde haya que transmitir / recibir / procesar datos (registros semánticos)

Registros semánticos

- Dada la simpleza de micro solo hay dos tipos de registros semánticos
  - REG_OPERACION, que solo contendrá el valor del token SUMA o RESTA.
  - REG_EXPRESION, que contendrá el tipo de expresión y el valor que le corresponde. Este valor puede ser:
    - una cadena (para el caso de un identificador)
    - un número entero (para el caso de una constante)

Máquina Virtual de micro

- Para Micro usaremos una MV con instrucciones del tipo:

  OP A,B,C

- **OP**: Código de Operación
- **A,B**: operandos
- **C**: donde almacenar el resultado
- Dependiendo de OP todos los demás operandos pueden estar o no
- Ejemplos
  - Declara dato,Entero
  - Resta dato,27,resultado (resultado := dato - 27)

Gramática con Anotaciones

<objetivo> → <programa> FDT **#terminar**

<programa> → **#comenzar** inicio <listaSentencias> fin

<listaSentencias> → <sentencia> {<sentencia>}

<sentencia> → <identificador> := <expresión> **#asignar** ; |
leer (<listaIdentificadores>); |
escribir (<listaExpresiones>);

<listaIdentificadores> → <identificador> **#leer_id** {, <identificador> **#leer_id**}

<listaExpresiones> → <expresión> **#escribir_exp** {, <expresión> **#escribir_exp**}

<expresión> → <primaria> {<operadorAditivo> <primaria> **#gen_infijo**}

<primaria> → <identificador> |
CONSTANTE **#procesar_cte** |
(<expresión>)

<operadorAditivo> → SUMA **#procesar _op** | RESTA **#procesar_op**

<identificador> → ID **#procesar_id**

Algunas rutinas semánticas

- Se agrega la producción <identificador> de modo tal de llamar a la rutina ProcesarID cada vez que se haga Match(ID)
- ProcesarID:
  - chequea el identificador, es decir, si no está en la Tabla de símbolos lo agrega
    - Genera el código de declaración de la variable.
  - Arma y devuelve el registro semántico del identificador
- ProximoToken: cuando pide un token al scanner lo busca en Tabla de símbolos. Esto permite corregir el token cuando es una palabra reservada (recordar que el scanner devuelve ID).

Ejemplo PAS modificada

- PAS original

```
void Expresion (void) {
// <expresion> -> <primaria> {<operadorAditivo> <primaria>}
    token t;
    Primaria();

    for (t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken()) {
        OperadorAditivo();
        Primaria();
    }
}
```

- Producción con anotaciones

<expresión> → <primaria> {<operadorAditivo> <primaria> **#gen_infijo**}

- Además de generar la operación debe manejar los registros semánticos

Ejemplo PAS modificada

- PAS con rutinas semánticas

```
void Expresion (REG_EXPRESION *resultado)
{
// <expresión> -> <primaria> {<operadorAditivo> <primaria> #gen_infijo}

    REG_EXPRESION operandoIzq, operandoDer;
    REG_OPERACION op;
    token t;

    Primaria(&operandoIzq);

    for (t = ProximoToken(); t == SUMA || t == RESTA; t = ProximoToken()) {
        OperadorAditivo(&op);
        Primaria(&operandoDer);

        // Probable generación de variables temporales
        operandoIzq = GenInfijo(operandoIzq, op, operandoDer);
    }

    *resultado = operandoIzq;
}
```

Generar infijo

- GenInfijo: Genera la instrucción para una operación infija y construye un registro semántico con el resultado
  - Va generando variables temporales según haga falta (es donde va a almacenar el resultado de la operación)
  - Según el registro de operación genera el código de operación del MV que corresponde
  - Si los registros semánticos son de un ID los chequea (incorpora a TS y genera instruccion de declaración)
  - Chequea la variable temporal (para declararla) y genera instruccion MV
  - Arma registro semántico con la variable temporal y lo devuelve como resultado

Ejemplo de compilación

```
inicio
 leer(a, b);          Declare a,Integer,
                      Read a,Integer,
                      Declare b,Integer,
                      Read b,Integer,
 a := 3 + b - 5;      Declare Temp&1,Integer,
                      ADD 3,b,Temp&1
                      Declare Temp&2,Integer,
                      SUBS Temp&1,5,Temp&2
                      Store Temp&2,a,
 escribir (a);        Write a,Integer,
fin                   Stop ,,
```

Ejemplo con paréntesis

```
inicio
 leer(a, b);          Declare a,Integer,
                      Read a,Integer,
                      Declare b,Integer,
                      Read b,Integer,
 a := 3 + (b - 5);    Declare Temp&1,Integer,
                      SUBS b,5,Temp&1
                      Declare Temp&2,Integer,
                      ADD 3,Temp&1,Temp&2
                      Store Temp&2,a,
 escribir (a);        Write a,Integer,
fin                   Stop ,,
```

Licencia