Bloque 2:Análisis

TEMA 3: ANALIZADOR SINTÁCTICO

Índice

1. Introducción
2. Papel del analizador sintáctico y tipos
3. Implementación del AS
4. Cambio en la gramática
5. Análisis descendente
    5.1. Descendente recursivo con *backtracking*
    5.2. Predictivo recursivo y no recursivo
6. Análisis ascendente
    6.1. Ascendente genérico
    6.2. Por precedencia de operadores
    6.3. LR
7. Utilización de generadores de analizadores sintáctico: Yacc
8. Tratamiento de errores
9. Tratamiento de errores en YACC
10. Utilización de generadores de analizadores sintáctico: CUP

Índice

1. Introducción
2. Papel del analizador sintáctico y tipos
3. Implementación del AS
4. Cambio en la gramática
5. Análisis descendente
    5.1. Descendente recursivo con *backtracking*
    5.2. Predictivo recursivo y no recursivo
6. Análisis ascendente
    6.1. Ascendente genérico
    6.2. Por precedencia de operadores
    6.3. LR
7. Utilización de generadores de analizadores sintáctico: Yacc
8. Tratamiento de errores
9. Tratamiento de errores en YACC
10. Utilización de generadores de analizadores sintáctico: CUP

Introducción

- Todo lenguaje de programación tiene estructura sintáctica.
- Se puede expresar la sintaxis con:
  - Gramática
  - Notación BNF ó BNFE

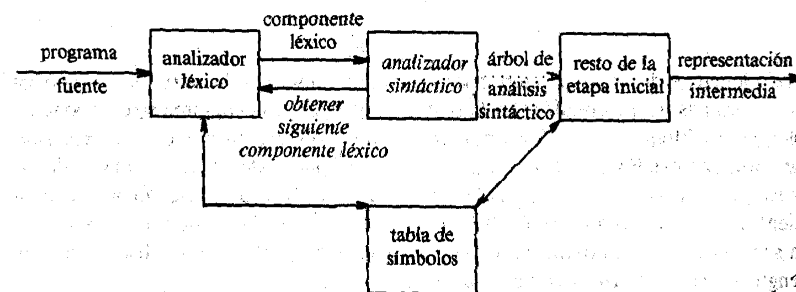
Introducción

- Ventajas del uso de gramáticas:
  - Proporciona una especificación sintáctica precisa
  - Construcción automática de analizadores sintáctico
  - Proporciona estructura a un lenguaje para la traducción a código objeto
  - Ayuda a la detección de errores
  - Permiten ampliar más fácilmente el lenguaje.

Índice

1. Introducción
2. Papel del analizador sintáctico y tipos
3. Implementación del AS
4. Cambio en la gramática
5. Análisis descendente
    5.1. Descendente recursivo con *backtracking*
    5.2. Predictivo recursivo y no recursivo
6. Análisis ascendente
    6.1. Ascendente genérico
    6.2. Por precedencia de operadores
    6.3. LR
7. Utilización de generadores de analizadores sintáctico: Yacc
8. Tratamiento de errores
9. Tratamiento de errores en YACC
10. Utilización de generadores de analizadores sintáctico: CUP

Papel del analizador sintáctico



Del libro "Compiladores. Principios, técnicas y herramientas" del autor Aho.

Papel del analizador sintáctico

- Se encarga de obtener una cadena de tokens.
- Comprueba si la cadena puede ser generada por la gramática del lenguaje fuente.
- Detecta errores e informa de ellos.
- Se recupera ante los errores para continuar compilando.

Papel del analizador sintáctico

- Tipos de analizadores sintácticos:
  - Descendentes: construyen el árbol sintáctico desde arriba hacia abajo (LL)
  - Ascendentes: comienzan en las hojas y suben hacia la raíz.
- Ambos examinan la entrada de izquierda a derecha.

Índice

1. Introducción
2. Papel del analizador sintáctico y tipos
3. Implementación del AS
4. Cambio en la gramática
5. Análisis descendente
    5.1. Descendente recursivo con *backtracking*
    5.2. Predictivo recursivo y no recursivo
6. Análisis ascendente
    6.1. Ascendente genérico
    6.2. Por precedencia de operadores
    6.3. LR
7. Utilización de generadores de analizadores sintáctico: Yacc
8. Tratamiento de errores
9. Tratamiento de errores en YACC
10. Utilización de generadores de analizadores sintáctico: CUP

Implementación AS

- Para implementar el AS se utilizan autómatas de pila.
  - Formados por:
    - Conjunto de estados
    - Alfabeto de entrada + alfabeto de pila
    - Transiciones (p,x,s;q,z)

Índice

1. Introducción
2. Papel del analizador sintáctico y tipos
3. Implementación del AS
4. Cambio en la gramática
5. Análisis descendente
   5.1. Descendente recursivo con *backtracking*
   5.2. Predictivo recursivo y no recursivo
6. Análisis ascendente
   6.1. Ascendente genérico
   6.2. Por precedencia de operadores
   6.3. LR
7. Utilización de generadores de analizadores sintáctico: Yacc
8. Tratamiento de errores
9. Tratamiento de errores en YACC
10. Utilización de generadores de analizadores sintáctico: CUP

Cambios en la gramática

- Existen técnicas para cambiar las gramáticas sin cambiar el lenguaje que generan.
  - Eliminar recursividad por la izquierda: consiste en transformar la recursividad por la izquierda en recursividad por la derecha
    - Ej:

      A→Ac|i (recursiva por la izquierda)

      A→iR; R→cR|lambda (recursiva por la derecha)

Cambios en la gramática

- Fatorización: se aplica cuando un no terminal puede derivar en dos cadenas que comiencen con el mismo terminal.
  - Ej:

    A→cBj|ckU (ambas producciones empiezan por c)

    U→gU

    B→b|bB (ambas producciones empiezan por b)

    A→cA'; A'→Bj|kU; U→gU; B→bB'; B'→bB'|lambda

Índice

1. Introducción
2. Papel del analizador sintáctico y tipos
3. Implementación del AS
4. Cambio en la gramática
5. Análisis descendente
   5.1. Descendente recursivo con backtracking
   5.2. Predictivo recursivo y no recursivo
6. Análisis ascendente
   6.1. Ascendente genérico
   6.2. Por precedencia de operadores
   6.3. LR
7. Utilización de generadores de analizadores sintáctico: Yacc
8. Tratamiento de errores
9. Tratamiento de errores en YACC
10. Utilización de generadores de analizadores sintáctico: CUP

Analizador sintáctico descendente

- Construyen el árbol buscando derivaciones a la izquierda hasta que concuerde con la entrada.
  - Derivación: sustituir un antecedente por un consecuente

Analizador sintáctico descendente

Tipos de analizadores descendentes

- Análisis sintáctico con *backtracking*:
  - Puede realizar retrocesos.
  - Ventajas:
    - Son los más potentes
  - Desventajas:
    - Analizan varias veces la entrada hasta encontrar las derivaciones correctas.
    - Son más lentos
    - Aceptan gramáticas ambiguas

    Ej.: S → cAd

    A → ab|a

    Entrada: cad

Analizador sintáctico descendente

Tipos de analizadores descendentes

- Analizadores sintácticos predictivos:
  - Intentan predecir la siguiente construcción.
  - Utilizan uno o más símbolos de anticipación.
  - Gramáticas LL (Left to Right, Leftmost derivation).
  - Si hay ambigüedad → no puedo aplicarlo
  - Dos tipos:
    - A. S. P. Recursivos: mediante llamadas recursivas
    - A. S. P. no recursivo: Con pila

Analizador sintáctico descendente

A.S.P. Recursivo

- No necesita retroceso
- Ideal para construir compiladores a mano
- Mantiene la pila de forma implícita mediante las llamadas recursivas.
- Características
  - NT→ implementan procedimientos
  - Cuerpo procedimiento: lo que especifique la regla

Analizador sintáctico descendente

- Cada procedimiento hace:
  - Decide la producción que utilizará analizando el símbolo de anticipación
  - Usa una producción imitando al lado derecho.
    - Un NT da como resultado una llamada al procedimiento del NT
    - Un T que coincida con el símbolo de anticipación da como resultado que se lea el componente léxico. Si el componente léxico de la producción no coincide con el símbolo de anticipación se declara un error.

Analizador sintáctico descendente

- Conjunto cabecera:
  - CAB(X)
  - Indica el primer símbolo de las cadenas derivadas e X.
  - Cálculo:

```
Si x pertenece a los terminales=> x pertenece a C
Si X-> lambda => CAB(X)+lambda
Si X es un no terminal y existe X->Y1Y2..Yn
=> CAB(X)+CAB(Y1)
CAB(X)+CAB(Yk) sii para todo i=1 hasta i=k-1
lambda pertenece a CAB(Yi)
```


Analizador sintáctico descendente

- Conjunto siguiente:
  - SIG(X). Representa los terminales inmediatamente a continuación de X en cualquier forma sentencial.
  - Cálculo:

```
Si X es el axioma ->añadimos $ (fin de fichero)
Si existe Y->aXb entonces SIG(X)+z para todo z
perteneciente a CAB(b), excepto lambda.
Si existe Y->aX ó existe Y->aXb y lambda
pertenece a CAB(b) entonces SIG(X)+SIG(Y)
```

Analizador sintáctico descendente

- Ejemplo: Dada la siguiente gramática, calcular los conjuntos cabeceras y siguientes.

```
E->TE'
E'->+TE'|lambda
T->FT'
T'->*FT'|lambda
F-> (E)|id
```


Analizador sintáctico descendente

- Solución:

```
PRIMERO (E)=PRIMERO(T)=PRIMERO(F)={(,id}
PRIMERO (E')={+,lambda}
PRIMERO (T')={*, lambda}
SIGUIENTE(E)=SIGUIENTE(E')={),$}
SIGUIENTE(T)=SIGUIENTE(T')={+,),$}
SIGUIENTE(F)={+,*,),$}
```

Analizador sintáctico descendente

- Conjunto cabecera de una producción:

```
CAB(Y->Xa)=
Si {Xa}={lambda} entonces SIG(Y)
sino
si X es terminal {X}
si X es no terminal
    Si CAB(X) no contiene lambda entonces CAB(X)
    sino CAB(X) U CAB(Y->a)
```


Analizador sintáctico descendente

- Condición LL(1):
  - Eliminar la recursividad por la izquierda
  - Un símbolo de anticipación es suficiente:
    - Sin producciones lambda:

      $CAB(A ::= \alpha j) \cap CAB(A ::= \alpha i) = \emptyset$
    - Con producciones lambda:

      $CAB(A ::= \alpha j).SIG(A) \cap CAB(A ::= \alpha i).SIG(A) = \emptyset$

Analizador sintáctico descendente

- Ejemplo: Dada la siguiente gramática, transformarla si es necesario y crear el código del ASP recursivo.

Exp→EXP addop term | term
addop→+|-
term →term mulop factor|factor
*mulop→**
factor→(exp)|number

Analizador sintáctico descendente

- Solución:
  - Quitar la recursividad

Exp→term exp'
Exp'→addop term exp'|lambda
addop→+|-
term→factor term'
term'→mulop factor term'|lambda
*mulop→**
factor→(exp) | number

Analizador sintáctico descendente

- Solución:
  - Factorizar: no hace falta
  - Conjuntos cabeceras:

```
CAB(EXP)=CAB(TERM)=CAB(FACTOR)={(,number} CAB(EXP')={lambda} U
CAB(ADDOP) = {lambda,+,-} CAB(ADDOP)={+,-} CAB(TERM')= {lambda} U
CAB(MULOP)={lambda,*} CAB(MULOP)={*}
```


Analizador sintáctico descendente

- Solución:
  - Conjuntos siguientes:

```
SIG(EXP)={$,)} SIG(EXP')=SIG(EXP)={$,)} ---(regla Y->aX; se ponen
los sig(y))--- SIG(ADDOP)=CAB(TERM)={(, number}
SIG(TERM)=CAB(EXP')-{lambda} U SIG(EXP') U SIG(EXP)={+,-,$,)}
SIG(TERM')=SIG(TERM)={+,-,$,)} SIG(MULOP)=CAB(FACTOR)={(,number}
SIG(FACTOR)=CAB(TERM')-{lambda} U SIG(TERM') U SIG(TERM)={*,+-$,)
```

Analizador sintáctico descendente

- Solución:
  - Conjunto cabecera de las producciones

```
CAB(exp->term exp') =CAB(term)={(,number} CAB(exp'->addop term
exp')=CAB(addop)={+,-} CAB(exp'->lambda)=SIG(exp')={$,)}
CAB(addop->+)={+} CAB(addop->-)={-} CAB(term->factor
term')=CAB(factor)={(,number} CAB(term'->mulop factor
term')=CAB(mulop)={*} CAB(term'->lambda)=SIG(term')={+,-$,)}
CAB(mulop->*)={*} CAB(factor->(exp))={(}
CAB(factor->number)={number}
```


Analizador sintáctico descendente

- Solución:
  - Condición LL(1): Sólo hay que ver que las cabeceras sean disjuntas.
  - Construcción del autómata:

```
void exp() {
    if(anticip=number || anticip='(')
    {
        term();
        exp'();
    }
    else error();
}
```

Analizador sintáctico descendente

```
void term() {

    if(anticip='(' || anticip=number) factor(); term'();
    else
    {
        error();
    }
```

```
void term'() {
    if(anticip='*')
    {
      mulop(); factor(); term'();
    }
    else
    {
        if(anticip!={+,-$,)} error();
    }
}
```


Analizador sintáctico descendente

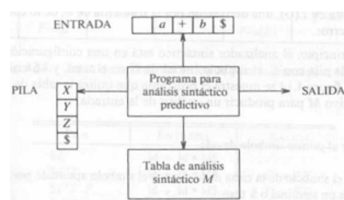
```
void mulop () {
    if(anticip='*')cmp(*);
    else error();
}


void factor() {
    switch(anticip)
    {
        case (: cmp((); exp(); cmp())
        case number: cmp(number)
        default: error();

    }
}
```

Analizador sintáctico descendente

A.S.P. no recursivo

- Se mantiene una pila de forma explícita
- Esta guiado por:
  - La cima de la pila
  - Tabla LL(1)
  - Entrada
- Según esto se decide si:
  - Aceptar
  - Pasar al siguiente símbolo
  - Aplicar producción
  - Notificar error

Analizador sintáctico descendente



La tabla: M[A,a] parte de la derecha de una producción
X es la cima de la pila
a es el carácter de entrada:

- Si X=a=$ ➔ aceptada la entrada.
- Si X=a ➔ saca X de la pila y avanza el puntero entrada.
- Si X es un no terminal ➔ M[X,a]
  - Si está vacío notificar error
  - Si tiene una producción, desapilar antecedente y aplicar consecuente

Analizador sintáctico descendente

- Para construir la tabla LL(1):
  - Filas: terminales (T)
  - Columnas: no terminales (NT)
  - Para todo A que sea NT y para toda producción A::=z:
    - Para todo a perteneciente a CAB(A→z); M[A,a]=A→z
    - Si lambda pertenece a CAB(A→z); M[A,b]=A→z para todo b perteneciente a SIG(A).
    - Si A es el último poner $

Analizador sintáctico descendente

Gramática de if then else:
P→iEtPP'|a
P'→ eP| λ
E→ b

P→iEtP|iEtPeP|a
E→b

Siguiente(P')=siguiente(P)
={$,Primero(P')}={$,e}

	a	b	e	i	t	$
P	P→a			P →iEtPP'		
P'			P'→eP P'→λ			P'→λ
E		E→b				

Analizador sintáctico descendente

Gramática de if then else:

P→iEtPP'|a
P'→ eP| λ
E→ b

P→iEtP|iEtPeP|a
E→b

Siguiente(P')=siguiente(P)
={$,Primero(P')}={$,e}

	a	b	e	i	t	$
P	P→a			P →iEtPP'		
P'			P'→eP P'→λ			P'→λ
E		E→b				

Gramáticas ambiguas generan tablas con entradas conflictivas: necesario gramáticas LL(1)

Índice

1. Introducción
2. Papel del analizador sintáctico y tipos
3. Implementación del AS
4. Cambio en la gramática
5. Análisis descendente
   - 5.1. Descendente recursivo con backtracking
   - 5.2. Predictivo recursivo y no recursivo
6. Análisis ascendente
   - 6.1. Ascendente genérico
   - 6.2. Por precedencia de operadores
   - 6.3. LR
7. Utilización de generadores de analizadores sintáctico: Yacc
8. Tratamiento de errores
9. Tratamiento de errores en YACC
10. Utilización de generadores de analizadores sintáctico: CUP

Analizador sintáctico ascendente

- Construyen el árbol desde abajo hacia arriba. Empiezan por los nodos hoja (formados por símbolos de la entrada) y avanzan hacia arriba hasta llegar al axioma (reducen la entrada al axioma).
  - Reducen una cadena w al axioma de la G.
  - La reducción: derivación en sentido inverso

Analizador sintáctico ascendente

- Hay tres tipos:
  - AS ascendente o AS por desplazamiento y reducción
    - Funcionamiento:
      - Reducir una cadena w al símbolo inicial de la gramática.
      - En cada paso de reducción se sustituye una subcadena determinada que concuerde con el lado derecho de una producción por el símbolo del lado izquierdo de dicha producción

Analizador sintáctico ascendente

- Ejemplo:

A->Abc|b

B->d

La cadena abbcde se puede reducir a S por los siguientes pasos: abbcde, aAbcde, aAde, aABe, S.

Analizador sintáctico ascendente

- Desventajas:
  - Muchas comprobaciones de la cadena
  - Comprobar la concordancia en la cima de la pila con las reglas puede ser lento.

Analizador sintáctico ascendente

- Análisis sintáctico por precedencia de operadores
  - Analizadores más sencillos.
  - Propiedades de las gramáticas:
    - Ningùn lado derecho de la producción es lambda
    - No existen dos no terminales adyacentes.
    - Si se cumplen estas propiedades, se denominan gramáticas de operadores.
  - Ejemplo: No es G de operador:

```
E->EAE| (E) |-E|id
A->+|-|*|/
```


Analizador sintáctico ascendente

- Relaciones de precedencias:
  - a<=b: a tiene menor precedencia que b
  - a=>b: a tiene mayor precedencia que b
  - a=b: a tiene la misma precedencia que b
- Significado de las relaciones:
  - a=>a quiere decir que se reduce antes que b por lo tanto está más abajo en el árbol.
- Objetivo de las relaciones: delimitar los consecuentes

Analizador sintáctico ascendente

- Diferencia con los operadores aritméticos:
  - Se puede tener a=>b y a<=b para el mismo lenguaje → ambigüedad
  - Esto no ocurre con los operadores aritméticos
- Resolución de ambigüedad:
  - Asociatividad: en el caso de operadores con la misma prioridad hay que asociarlos por la izquierda o por la derecha
  - Precedencia de operadores: poner los operadores más prioritarios en la parte más baja del árbol
    - Ejemplo: Si *=>+ se hace +<=* y *>=+

Analizador sintáctico ascendente

- Inconvenientes:
  - Componentes léxicos con dos precedencias. Resolver la ambigüedad puede ser complejo.
  - La relación entre el analizador y la G es muy débil → no seguro reconocer L(G)
  - Pocas gramáticas de operador

Analizador sintáctico ascendente

- Analizadores sintácticos LR(K):
  - L: examen de izquierda. a derecha.
  - R: construir derivación por la derecha (de der. a izq.)
  - K: nº de símbolos. de anticipación

Analizador sintáctico ascendente

- Ventajas:
  - Reconoce muchos L de G. Independientes de contexto
  - Desplazamiento y reducción sin retroceso
  - Encuentra el error tan pronto como es posible en examen de izq. a der. Se pueden construir analizadores LR para reconocer prácticamente todas las construcciones de los lenguajes de programación
  - Es el método más genérico que se conoce y es tan eficiente como los otros
  - Reconoce más gramáticas que los analizadores predictivos

Analizador sintáctico ascendente

- Inconveniente:
  - Costoso su diseño para lenguajes de programación → usar generadores automáticos: YACC
- Existen tres técnicas para construir una tabla de analizador LR:
  - LR sencillo, SRL: es el más fácil de implantar, pero el menos poderoso
  - LR canónico, LR(1): es más poderoso que el SRL y más complejo
  - LALR: está entre los dos anteriores en complejidad. Funciona con la gramática de la mayoría de los lenguajes de programación.

Analizador sintáctico ascendente

- Funcionamiento:
  - Pila con marcas (estado)
  - Tabla A. Sintáctico: acción, ir_a
  - Configuración (pila, entrada):

    (s0 X1s1 X2s2...XmSm, ai ai+1....an$)

1) Si acción[sm,ai]=desplazar, s

    (s0 X1s1 X2s2...XmSm ais, ai+1....an$)

2) Si acción[sm,ai]= reducir A→B, ir_a

    (s0 X1s1 X2s2...Xm-rSm-r As, ai ai+1....an$)

donde ir_a[sm-r, A]=s y r es la longitud de B

Se realiza la acción semántica asociada

3) Si acción[sm,ai]= aceptar (finalizar éxito)

4) Si acción[sm,ai]= error (llamar a recuperación de errores)

Índice

1. Introducción
2. Papel del analizador sintáctico y tipos
3. Implementación del AS
4. Cambio en la gramática
5. Análisis descendente
    5.1. Descendente recursivo con backtracking
    5.2. Predictivo recursivo y no recursivo
6. Análisis ascendente
    6.1. Ascendente genérico
    6.2. Por precedencia de operadores
    6.3. LR
7. Utilización de generadores de analizadores sintáctico: Yacc
8. Tratamiento de errores
9. Tratamiento de errores en YACC
10. Utilización de generadores de analizadores sintáctico: CUP

Generación automática de AS, YACC

- Características:
  - Generador automático de AS, que genera un autómata LALR.
  - Genera código C
  - Trabaja con el generador de AL LEX.
  - A partir de una especificación en un fichero .y (reglas) genera un fichero .c que contiene la implementación del algoritmo LR y la tabla LALR

Generación automática de AS, YACC

- Un programa fuente en YACC tiene tres partes:

```
%{
        declaraciones en C
%}
        declaraciones de símbolos
%%
          producciones de la gramática
%%
código del usuario (funciones y procedimientos)
```


Generación automática de AS, YACC

- En la sección declaraciones C:
  - Declaraciones ordinarias de C
    - Variables temporales
- En la sección de declaración de símbolos:
  - Declaración de componentes léxicos
    - Ejemplo: %token Dígito
  - La precedencia va de menor a mayor de arriba abajo.
    - La forma de escribir los tokens determina la precedencia
  - Se definen los símbolos no terminales
  - Se agrupan los elementos por su precedencia y asociatividad

Generación automática de AS, YACC

- Sección de producciones:
  - Contiene las reglas de traducción.
    - Cada regla consta de:
      - Una producción de la gramática
      - La acción semántica asociada
      - Ejemplo:
        - Regla: <lado izquierdo> → <alt1>|<alt2>|<alt3>
        - Yacc:

          <lado izquierdo> : <alt1> {acción semántica 1} |
          <alt2> {acción semántica 2}…

Generación automática de AS, YACC

- Sección de código de usuario:
  - Contiene rutinas de apoyo escritas en C.
  - Se debe proporcionar un análisis léxico con nombre yylex().
  - Se pueden agregar rutinas de recuperación de errores.

Generación automática de AS, YACC

- Definición de tokens:
  - Un carácter simple ‘c’ se considera como Terminal
  - Las cadenas sin declarar se consideran no terminales
  - El axioma se declara en la sección de declaración con %start
    - Si no se especifica se toma el primer símbolo
  - Los tokens se declaran en la sección de declaración con %token.
    - %Left idTokens: tokens con asociatividad por la izquierda
    - %Right idTokens: tokens con asociatividad por la derecha
  - El resto de símbolos que aparecen son NT

Generación automática de AS, YACC

- Precedencia y asociatividad:
  - La precedencia que se le asigna a la regla es la del símbolo terminal de más a la derecha.

  Ej.: Exp: exp ’+' exp
  - Se puede modificar la precedencia de la regla mediante el modificador %prec SIMBOLO, teniendo la precedencia de SIMBOLO:

  Ej.: Exp: ’-' exp %prec MENOSU
  - Se puede indicar que no haya asociatividad (rompera asociativaidad) mediante: %nonasoc

Generación automática de AS, YACC

- %left ‘+’ ‘-’: + y – tienen la misma precedencia y tienen asociatividad por la izquierda
- %right ‘!’: asociativo por la derecha y mayor precedencia que los anteriores por estar más abajo
- %nonasoc ‘!’: obliga al operador a ser unario
- %prec: sirve para modificar la precedencia de una regla

Generación automática de AS, YACC

- Ejecución de las acciones:
  - Las acciones semánticas se ejecutan al aplicar la reducción.
  - Existe una pila de valores y otra de tokens

Generación automática de AS, YACC

- Comportamiento de las pilas para la entrada 12+15

entrada	p.valores	p.tokens
12	$1=12	NUMERO
12+		NUMERO +
12+15	$3=15	NUMERO + NUMERO
-	-	reduce: exp
		(exp + exp)
		$1 $2 $3

Generación automática de AS, YACC

- Cuando lex encuentra un token:
  - Devuelve token → en la pila de token
  - Devuelve valor en yylval → en la pila de valores o atributos
  - A los elementos de la pila de valores se le asocia $i, siendo i la posición de su token en la parte derecha de la regla
  - El NT de la parte izq. de la regla es $$
  - La acción por defecto (siempre realiza) es $$=$1

Generación automática de AS, YACC

- Ejemplo: una calculadora que realiza las cuatro funciones básicas aritméticas

Generación automática de AS, YACC

```
%{
#define YYSTYPE double  /* define la pila*/
%}
%token NUMERO
%left '+' '-'
%left '*' '/'
%left MENOSU
%%
exp: NUMERO       {$$=$1;}
        | '-' exp  %prec MENOSU {$$=$1;}
        | exp '+' exp { $$=$1+$3; }
        | exp '-' exp { $$=$1-$3; }
        | exp '*' exp { $$=$1*$3; }
        | exp '/' exp { $$=$1/$3; }
        | '(' exp ')'   { $$=$2; }
        ;
%%
        En algún sitio: int yylval;
```

Generación automática de AS, YACC

- Estructuración de ficheros:
  - Fuente.y: la especificación de la gramática
  - Yacc genera:

yytab.h:

```
#define TOKEN1 250
#define TOKEN2 251
...
typedef unio{
      char *nombre;
      int valor;
      ....
}YYSTYPE
extern YYSTYPE yylval;    /*si he declarado la pila en el
   fuente de yacc*/
```


Generación automática de AS, YACC

- Fuente.c: contiene la implementación del autómata.
- Debe haber un fichero con la función principal para iniciar el análisis:

```
int main (void){
   ....
   yyparse();
   ....
}
```

donde yyparse lo define yacc así:

```
yyparse(){
while (!eof (yyin))
   yylex();
.....}
```

Generación automática de AS, YACC

Resolución de ambigüedades

- Tipos de ambigüedades:
  - Desplazamiento/Reducción (shift/reduce):
    - Ejemplo:

exp: exp '+' exp
|exp '*' exp
|NUM
Entrada: 2+3*5
Pila: 2, 2+, 2+3, puedo
Puedo reducir: exp ´+´exp
Puedo desplazar: 2+3*

Generación automática de AS, YACC

Resolución de ambigüedades

- Reducción/Reducción (reduce/reduce):
  - Ejemplo:

exp: exp '+' exp '+'exp
| exp '+' exp
Entrada 2+3+4
y en la pila tengo:
(base pila)2,2+,2+3,2+3+,2+3+4 (cima de la pila)
Puedo reducir: exp '+' exp '+'exp
Puedo reducir: exp '+' exp

Generación automática de AS, YACC

Resolución de ambigüedades

- Se aplican las siguientes reglas:
  - Prioridad token de entrada > regla ➔ shift
  - Prioridad token de entrada < regla ➔ reduce
  - Prioridad token de entrada = regla ➔ utiliza asociatividad:
    - left ➔ reduce
    - right ➔ shift
    - nonasoc ➔ error

Generación automática de AS, YACC

Resolución de ambigüedades

- Si no hay especificado prioridades ni asociatividades:
  - ante un shift/reduce aplica un shift

    (precedencia a la cadena más larga).
  - ante un reduce/reduce aplica el reduce de la primera producción escrita en la gramática

    (importante el orden en que se escriben las producciones)

Generación automática de AS, YACC

- Ejemplo:
  - Describir una gramática que genera expresiones aritméticas y relacionales y asignaciones (en las expresiones se pueden utilizar la sentencia IF).

Generación automática de AS, YACC

Solución

```
%{
#include <stido.h>
...
%}
%start list
%token IF ELSE
%token ID NUM
%nonasoc IGUAL '<' '>'
%left '+' '-'
%left '*' '/'
%left UMINOS
%right '^'
%%
list:
|list stat

stat: exp '\n'
| ID '=' exp '\n‘
```

Generación automática de AS, YACC

Solución

```
exp: '(' exp ')'
| exp IGUAL exp
| exp '<' exp
| exp '>' exp
| exp '+' exp
| exp '-' exp
| exp '*' exp
| exp '/' exp
| '-' exp   %prec UMINOS
| '+' exp   %prec UMINOS
| exp '^' exp
|IF '(' exp ') exp
|IF '(' exp ') exp ELSE exp
|ID
|NUM
%%
```


Índice

1. Introducción
2. Papel del analizador sintáctico y tipos
3. Implementación del AS
4. Cambio en la gramática
5. Análisis descendente
    5.1. Descendente recursivo con backtracking
    5.2. Predictivo recursivo y no recursivo
6. Análisis ascendente
    6.1. Ascendente genérico
    6.2. Por precedencia de operadores
    6.3. LR
7. Utilización de generadores de analizadores sintáctico: Yacc
8. Tratamiento de errores
9. Tratamiento de errores en YACC
10. Utilización de generadores de analizadores sintáctico: CUP

Manejo de errores

- Tipos de errores que suelen ocurrir (dependiendo de la fase):
  - Léxicos: no concuerda con ninguna ER.
    - Ejemplo: escribir mal una palabra clave
  - Sintácticos: la estructura que se ha seguido no es correcta.
    - Ejemplo: expresión con paréntesis no emparejados
  - Semánticos: la estructura está bien pero hay errores de significado
    - Ejemplo: operador y operandos incompatibles.
  - Lógicos: los comete el programador
    - Ejemplo: una llamada infinitamente recursiva

Manejo de errores

- Algunos errores se pueden detectar en compilación otros solo en ejecución.
- El tratamiento de errores es una parte importante que se suele descuidar

Manejo de errores

- Gran parte de la detección y recuperación se centra en el AS:
  - Muchos errores de naturaleza sintáctica
- Recuperación: al producirse un error el compilador debe ser capaz de informar del error y seguir compilando. (ideal)

Manejo de errores

- Objetivos del manejo de errores:
  - Informar de la presencia de un error
  - Recuperarse rápido para detectar el mayor número de errores posteriores.
  - No retrasar la compilación de programas correctos

Manejo de errores

- Una vez que se detecta un error, ¿qué se hace?
- Tipos de estrategia de tratamiento y recuperación:
  - Modo pánico
  - Recuperación a nivel de frase
  - Producciones de error
  - Correcciones a nivel global

Manejo de errores

- Modo pánico:
  - Desecha símbolos de la entrada hasta encontrar un token de sincronización.
    - Tokens de sincronización: conjunto de tokens significativos que terminan bloques de código (delimitadores de secuencias: ;, palabras clave end).

Manejo de errores

- Modo pánico:
  - Ventaja:
    - Fácil de implementar
    - No cae en lazos infinitos.
  - Desventaja:
    - Puede pasar mucha entrada sin comprobar → ignorar otros errores.

Manejo de errores

- Recuperación a nivel de frase:
  - Se basa en la corrección local de la entrada
  - Sustituir una parte de la entrada restante (un prefijo) por alguna cadena que le permita continuar el análisis
    - Ej. de corrección local: sustituir una coma por un ; o suprimir un ; sobrante o insertar un ; que falte.

Manejo de errores

- Recuperación a nivel de frase:
  - Complicado:
    - Si no se escoge la corrección local correcta → lazos infinitos.
  - Desventaja:
    - No se recupera bien cuando el error se produce antes de donde se detecta

Manejo de errores

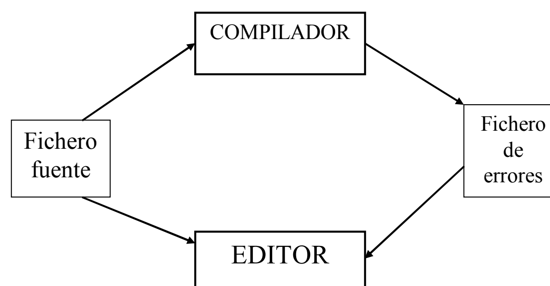
- Producciones de error:
  - Se amplía la gramática con producciones que generan construcciones erróneas.
  - Se asocia a dichas producciones diagnóstico y acciones de error

Manejo de errores

- Corrección global:
  - Algoritmos que determinan una secuencia mínima de cambios para obtener una corrección global de menor coste.
  - Pasan de una entrada incorrecta x a otra y deducida por la gramática.
  - Desventaja:
    - Costoso en tiempo y espacio ➔ solo se usa en teoría.

Manejo de errores

- Hay que tratar el error y visualizar su información:



Índice

1. Introducción
2. Papel del analizador sintáctico y tipos
3. Implementación del AS
4. Cambio en la gramática
5. Análisis descendente
    5.1. Descendente recursivo con backtracking
    5.2. Predictivo recursivo y no recursivo
6. Análisis ascendente
    6.1. Ascendente genérico
    6.2. Por precedencia de operadores
    6.3. LR
7. Utilización de generadores de analizadores sintáctico: Yacc
8. Tratamiento de errores
9. Tratamiento de errores en YACC
10. Utilización de generadores de analizadores sintáctico: CUP

Manejo de errores en YACC

- En YACC se realiza una recuperación de errores usando producciones que de error
    - Ejemplo:

      exp: exp '+' exp
        | exp '+' '+'exp {tratamiento del error}
        |NUM

Manejo de errores en YACC

- La recuperación del error se basa en tres elementos:
  - La función **yyerror()**
  - El token **error**
  - La acción predefinida **yyerrork**

Manejo de errores en YACC

- La función **yyerror()**:
  - Se encarga de mostrar información del error.
  - void yyerror(const char *msg)
  - Yacc necesita que el usuario suministre la función yyerror();
  - Cada vez que se produce un error, el parser llama a esta y le pasa un mensaje (suele ser "syntax error").

Manejo de errores en YACC

- La función **yyerror()**:

  La definición suele ser:

```
void yyerror(const char *msg){
  fprintf(stderr,"Error: %s\n", msg);
}
```

  - Si queremos más información:
    - Información que mantiene el léxico:

      yylineno, yycolumn, yycaract, yyfuente
    - Conjunto primeros del NT para decir lo que esperaba (yacc genera ficheros con información sobre tokens)

Manejo de errores en YACC

- El token **error:**
  - Palabra reservada que Yacc trata como un símbolo terminal. No lo devuelve el léxico.
  - Un token más para escribir producciones que describen situaciones de error.
  - Cuando no hay transición lo genera y trata de continuar el análisis con él

```
exp: exp '+' exp
   | NUM
   | error
```

Manejo de errores en YACC

- El token **error:**
  - Se coloca en NT importantes para la recuperación
    - exp, secuencias, bloques y procedimientos
  - Al detectar error:
    - Extrae de la pila hasta llegar un estado que pueda desplazar error a la pila

Manejo de errores en YACC

- El token **error:**
  - Al detectar error:
    - Si b es palabra vacía: se reduce y se invoca la acción de recuperación del usuario
    - Si b no es palabra vacía:
      - salta entrada para reducir b (si es terminales, desplaza)
      - Cuando tiene en la pila error b, reduce

Manejo de errores en YACC

- Ej.

```
Lineas : lineas expr '\n'
        | lineas '\n'
        |
        |error '\n' {tratamiento rutina}
```


Manejo de errores en YACC

- El token **error**

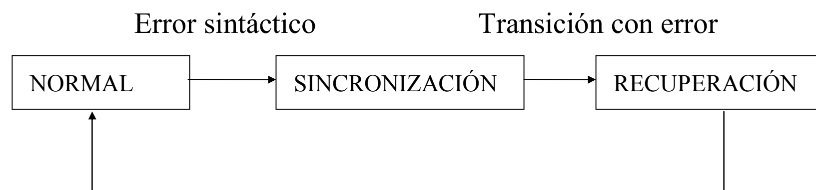
  Yacc lo usa de dos posibles formas:

  ej: 1++2

  - Token ficticio para reemplazar al que falta (genera error y desplaza)
  - Token para ignorar parte de la entrada (genera error, reduce por error y avanza un token la entrada).

Manejo de errores en YACC

- La acción predefinida **yyerrork**:
  - Cuando se ejecuta esta acción obliga al parser a volver al estado Normal



Manejo de errores en YACC

- La acción predefinida **yyerrork**:
  - Los estados en los que puede estar el parser son:
    - **NORMAL**: Modo habitual de funcionamiento. No hay error. Si se detecta error se ejecuta yyerror()
    - **SINCRONIZACIÓN**: desapilar símbolos de la pila hasta que consigue llegar a un estado de la cima el cual admita transición con el token error
    - **RECUPERACIÓN**: no se realiza tratamiento de nuevos errores. Se está en este estado hasta que se leen de la entrada **tres** tokens correctos. (evita encadenamiento de errores)

Manejo de errores en YACC

- El token error:
  - Hay una heurística para colocarlo:
    - Axioma: evita el desbordamiento de la pila
    - Símbolos de terminación de sucesiones ; , end. Conviene hacer yyerrok para evitar pérdida de detección (estado RECUPERACIÓN).
    - Construcciones recursivas, para que un error no descarte toda la lista

Manejo de errores en YACC

- Colocación de la acción yyerrok:
  - Detrás de lo símbolos que sirven como delimitadores de secciones.
    - Por ej. cierre de paréntesis, ';', 'end', etc. Se puede ejecutar siempre que reconozcamos uno de estos tokens.

      Ej.
      ```
      p_ce: ')'    { yyerrok; }
      p_c: ';'     { yyerrok; }
      end: END     { yyerrok; }
      ```

Manejo de errores en YACC

- Sitios donde normalmente se comenten errores:
  - En las comas, cierre de paréntesis, operadores en expresiones binarias, etc.).
  - Para saltarse la línea entera donde está el error:

```
start: exp '\n'
|ID '=' exp '\n'
|error '\n'    { yyerrok; }
```


Índice

1. Introducción
2. Papel del analizador sintáctico y tipos
3. Implementación del AS
4. Cambio en la gramática
5. Análisis descendente
   5.1. Descendente recursivo con backtracking
   5.2. Predictivo recursivo y no recursivo
6. Análisis ascendente
   6.1. Ascendente genérico
   6.2. Por precedencia de operadores
   6.3. LR
7. Utilización de generadores de analizadores sintáctico: Yacc
8. Tratamiento de errores
9. Tratamiento de errores en YACC
10. Utilización de generadores de analizadores sintáctico: CUP

Generación automática de AS, CUP

- CUP es otro generador de analizadores sintácticos LALR.
- El funcionamiento de CUP es muy parecido a YACC, pero más sencillo.
  - Está escrito en Java y genera código Java
  - Trabaja con JFLEX
  - Manual: http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html

Generación automática de AS, CUP

- Una especificación en CUP tiene 4 partes
  - 8 secciones específicas, la mayor parte de ellas opcionales
- Una especificación consiste:
  - Sección de especificación de paquetes e *imports*
  - Sección de código de usuario
  - Sección de lista de símbolos: terminales y no terminales
  - Sección de declaraciones de precedencia
  - Sección de declaración de gramática

Generación automática de AS, CUP

- Sección de especificación de paquetes e *imports*
  - Una especificación comienza con la declaración opcional del paquete
  - Tiene el mismo significado que en Java
  - También se pueden hacer los imports necesario

Generación automática de AS, CUP

- Sección de código de usuario:
  - En esta sección se puede incluir el código que el usuario quiera añadir al parser.
  - El código que se ponga aquí se copia directamente al parser.
    - Parser code{:….:}

Generación automática de AS, CUP

- Sección de lista de símbolos:
  - Primera sección de declaración obligatoria
  - Se especifican los símbolos:
    - Terminales:
      - terminal nombre1, nombre2…;
    - No terminales
      - Nonterminal nombre1, nombre2…;

Generación automática de AS, CUP

- Sección de declaración de precedencias:
  - Sección para tratar las ambigüedades de las gramáticas
  - Existen tres tipos de precedencias/asociatividades
    - precedence left *terminal*[, *terminal*...];
    - recedence right *terminal*[, *terminal*...];
    - precedence nonassoc *terminal*[, *terminal*...];

Generación automática de AS, CUP

- El orden de la precedencia va de abajo a arriba. Cuanto más abajo mayor precedencia.
  - precedence left ADD, SUBTRACT;
  - precedence left TIMES, DIVIDE;
    - La multiplicación y la división tienen mayor precedencia que la suma y la resta

Generación automática de AS, CUP

- Los terminales que no tengan asignada una precedencia de forma explícita tendrán menor precedencia que los que están en la lista.
- La precedencia de la producción viene dada por la precedencia de su último terminal. Si no tiene terminales tiene aún menor precedencia.
  - Ejemplo: expr ::= expr TIMES expr tiene la misma precedencia que TIMES.

Generación automática de AS, CUP

- A cada terminal se le asocia una asociatividad. Hay tres tipos:
  - Left
  - right
  - nonassoc.
- Todos los terminales que no se declaran en esta sección tiene menor precedencia

Generación automática de AS, CUP

- Sección de declaración de la gramática:
  - Sección final de una declaración en CUP
    - Comienza con la declaración opcional de "start" (axioma).
    - Si no se declara el axioma se tomará como tal la parte izquierda de la primera regla declarada
  - Cada producción de la gramática tiene un lado izquierdo seguido del símbolo "::=" seguida de las acciones pertinentes
    - Para especificar la palabra vacía se deja un espacio.
    - La última producción debe ir seguida de ;
  - Cada símbolo de la parte derecha puede ser etiquetado opcionalmente. Las etiquetas aparecen después del símbolo, se indica con ":"
    - Los nombres de las etiquetas deben ser únicos en una producción.
  - Los lados derechos se separan con |.
  - El código Java aparece encerrado entre {::}. Este código se copia directamente al parser.
    - El valor que devuelve la regla debe introducirse en la variable RESULT.

Generación automática de AS, CUP

- Ejemplo:

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE, MOD; precedence
left UMINUS;
expr ::= MINUS expr:e {: RESULT = new Integer(0 -
e.intValue()); :} %prec UMINUS
```


Generación automática de AS, CUP

- Ejecución de CUP:
  - Para ejecutarlo hay que introducir la siguiente línea:
    - java java_cup.Main *options* < *inputfile*
    - Inputfile es el fichero que contiene la especificación de la gramática.
    - Existen una serie de opciones. Consultar el manual.

Generación automática de AS, CUP

- **Personalización del Analizador**
  - Cada parser está formado de tres clases.
    - La clase SYM. Contiene las constantes de cada terminal. Si se usa la opción de -nonterms también se incluyen los NT.
    - La clase PARSER. Contiene la definición de dos clases
      - La clase pública parser que implementa el parser actual
      - La clase no pública CUP$action que contiene el código de usuario y el código de declaración. También contiene un método CUP$do_action que guía la ejecución . En general todos los nombres que comiencen por CUP$ los usa CUP internamente. Esta clase proporciona tres tablas:
        - Tabla de producciones
        - Tabla de acción
        - Tabla de reducción-ir a

Generación automática de AS, CUP

- El nombre de cada una de estas clases de puede cambiar al que el usuario desee.
- Para esto hay que utilizar las opciones que da CUP al generar el parser.

Generación automática de AS, CUP

- Integración del Analizador Léxico.
  - Es muy sencillo incorporar un AL generado con JFLEX.
- Para introducir el AL éste debe implementar la interfac java_cup.runtime.Scanner.
- El parser generado tiene un constructor que toma como parámetro un AL
  - Éste será utilizado para obtener la cadena de tokens
  - La comunicación entre AL y AS es mediante subrutina
    - El AL es una subrutina del AS.

Generación automática de AS, CUP

- Recuperación de errores:
  - El sistema de recuperación de errores es igual que el que tiene YACC.
  - Existe un token especial llamado error.
  - Este token sólo entra en juego si se detecta un error sintáctico. En este caso el parser intenta reemplazar alguna parte de la cadena de entrada por el token error.

Generación automática de AS, CUP

- Por ejemplo:

  stmt ::= expr SEMI | while_stmt SEMI | if_stmt SEMI | ... |
  error SEMI ;

  - Esto indica que si no se puede aplicar ninguna de las producciones para stmt, entonces se debería lanzar un error sintáctico y la recuperación debería hacerse saltándose los tokens erróneos, lo que equivale a reemplazarlos por error hasta el punto en el que el parser pueda continuar con un semicolon.
  - Se considera que nos hemos recuperado de un error, cuando se reconoce un número determinado de tokens pasado el token error.
    - Se especifica con el método error_sync_size()
    - Por defecto es 3.

Generación automática de AS, CUP

- Funcionamiento:
  - El parser primero mira los estados en la cima de la pila que tienen transiciones con error.
  - Entonces empezará a desapilar estados e ignorar tokens hasta que pueda continuar el escaneo.
  - Después de descartar cada token el parser intentará continuar sin ejecutar la acciones semánticas.
  - Si el parser consigue analizar satisfactoriamente el número mínimo de tokens, entonces vuelve hacia atrás en la entrada y sigue la ejecución de manera normal.
  - Si llega al final sin conseguir recuperarse de un error, se dice que la recuperación ha fallado.