Traducción dirigida por sintaxis

Ing. Max Cerna

Agenda

1. Gestión de errores
2. AST
3. Recursive Descent Parsing
4. Recursive Descent Algorithm
5. Recursión por la izquierda

Gestión de errores

GESTIÓN DE ERRORES

- El propósito del compilador es
  - Para detectar programas no válidos
  - Traducir los válidos
- Muchos tipos de posibles errores (por ejemplo, en C)

Tipo	Ejemplo	Detectado por
Léxico	$	Lexer
Sintáctico	x * %	Parser
Semántico	int x; y = x(3);	Type Checker
Exactitud	tu prox. proyecto de compi	Usuario

GESTIÓN DE ERRORES

- El manejador de errores debe:
  - Reportar los errores de manera precisa y clara.
  - Recuperarse rápidamente de un error.
  - No ralentizar la compilación de código válido.

- Un buen manejo de errores no es fácil de lograr.

GESTIÓN DE ERRORES

- Enfoques de simples a complejos:
  - Modo pánico
  - Producción de errores
  - Corrección automática local o global

- No todos son compatibles con todos los generadores de analizadores.

GESTIÓN DE ERRORES

Modo pánico

El modo de pánico es el método más simple y popular

Cuando se detecta un error:

- Descartar tokens hasta que se encuentre uno con un rol claro
- Continuar desde allí

Buscando tokens de sincronización

- Por lo general, los terminadores de declaraciones o expresiones

GESTIÓN DE ERRORES

Modo pánico

Considere la expresión errónea

```
(1 + +2) + 3
```

Recuperación en modo pánico: Saltar al siguiente entero y luego continuar

Bison/Cup: use el error de terminal especial para describir cuanto saltar en la entrada

```
E → int|E + E|(E)|error int|(error)
```

GESTIÓN DE ERRORES

Modo pánico

Ejemplo, entrada:

int 123a = 5;

float x = 3.14.15;

string name = "Hello World;

GESTIÓN DE ERRORES

Modo pánico

Lexer, macros:

letter = [a-zA-Z]

digit = [0-9]

oper = [+-*/]

whitespace = [\t\n\r]

separator = [;()"]

GESTIÓN DE ERRORES

Modo pánico

Lexer, expresiones regulares:

id = letter (digit | letter)*

num = digit+ (.digit+)?

no_recognized =[^a-zA-Z0-9+\-*/ \t\n\r;()"]

malformed = digit+ letter+ digit* | digit* .[^digit]+

GESTIÓN DE ERRORES

Modo pánico

Parser, producciones SIN manejo de errores:

S -> ASSIGN;

ASSIGN -> id = EXPR

EXPR -> EXPR + TERM | EXPR - TERM | TERM

TERM -> TERM * FACTOR | TERM / FACTOR | FACTOR

FACTOR -> (EXPR) | num | id

GESTIÓN DE ERRORES

Modo pánico

Parser, producciones **CON** manejo de errores:

S -> ASSIGN; | *error ';'*

ASSIGN -> id = EXPR | *error '=' EXPR*

EXPR -> EXPR + TERM | EXPR - TERM | TERM | *error ('+' | '-')*

TERM -> TERM * FACTOR | TERM / FACTOR | FACTOR | *error ('*' | '/')*

FACTOR -> (EXPR) | num | id | *error ('(' | ')' | num | id)*

GESTIÓN DE ERRORES

Producciones de error

Especificar errores comunes conocidos en la gramática

Ejemplo:

- Escribe `5 x` en lugar de `5 * x`
- Añadir la producción `E → E E`

Desventaja:

- Complica la gramática

GESTIÓN DE ERRORES

Producciones de error

Otros ejemplos:

expresiones donde falta un paréntesis de cierre, como en ***5 * (3 + 2***

Añadir la producción E → (E

capturar errores donde un operador es repetido innecesariamente, como en 5 ++ 3

Añadir la producción E → E ++ E

GESTIÓN DE ERRORES

Corrección automática local o global

- Idea: encontrar un programa “cercano” correcto
  - Pruebe las inserciones y eliminaciones de tokens
  - Búsqueda exhaustiva

- Desventajas:
  - Difícil de implementar
  - Ralentiza el análisis de los programas correctos
  - “Cercano“ no es necesariamente el programa “previsto“

GESTIÓN DE ERRORES

- Pasado
  - Ciclo de recompilación lento (incluso una vez al día)
  - Encuentra tantos errores en un ciclo como sea posible

- Presente
  - Ciclo de recopilación rápida
  - Los usuarios tienden a corregir un error/ciclo
  - La recuperación de errores complejos es menos convincente

AST
(Abstract Syntax Trees)

AST

Un parser rastrea la derivación de una secuencia de tokens.

Pero el resto del compilador necesita una representación estructural del programa.

Árboles de sintaxis abstracta (Abstract Syntax Trees)

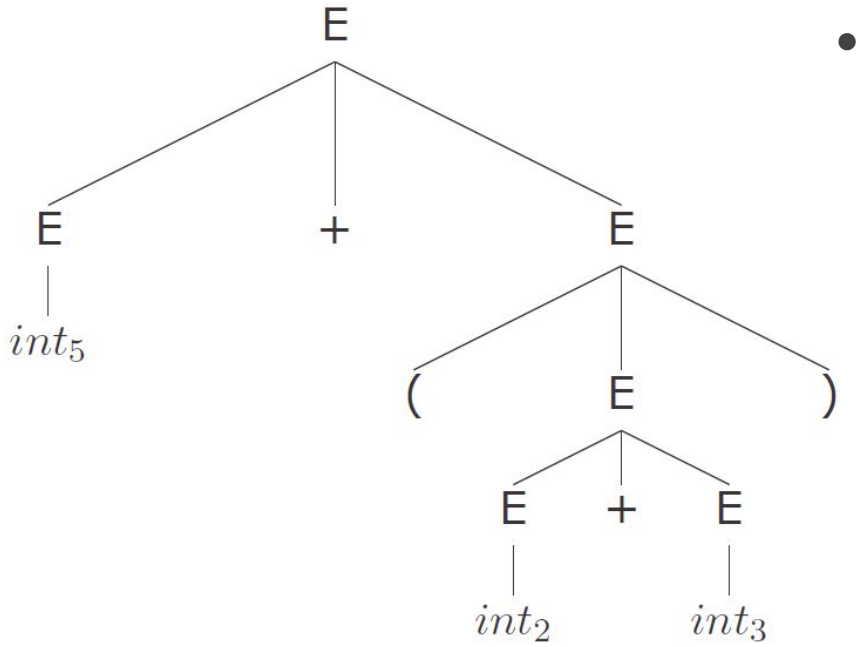
- Como los parse trees vistos hasta ahora pero ignorando ciertos detalles.
- Abreviado como AST

AST

- Considere la gramática
  - `E → int|(E)|E + E`
- Y la cadena
  - `5 + (2 + 3)`
- Después del análisis léxico (una lista de tokens)
  - `int5 + ( int2 + int3 )`

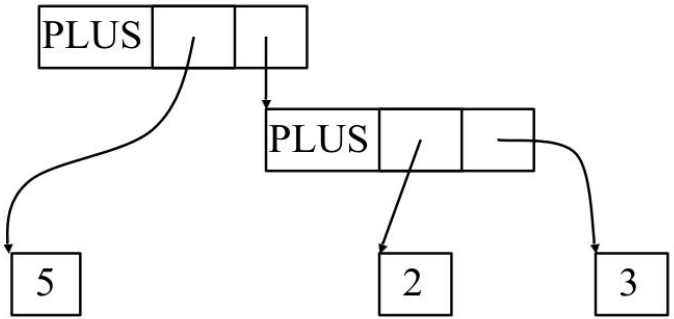
Ejemplo de Parse Tree

Durante el análisis construimos un árbol de análisis



- Un árbol de análisis
  - Rastrea el funcionamiento del parser.
  - Captura la estructura anidada
  - Mucha información (paréntesis, sucesores simples)

Ejemplo de Abstract Syntax Tree

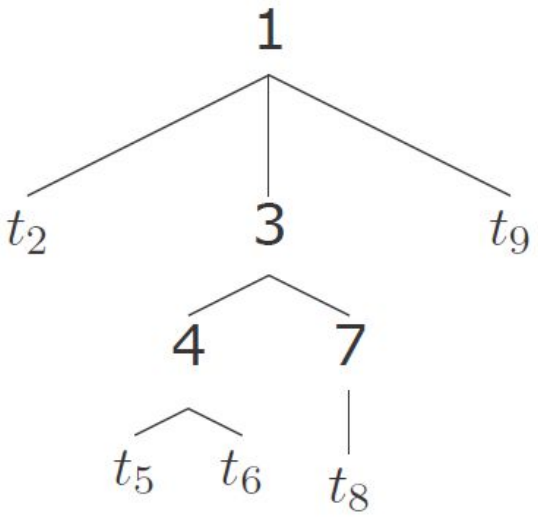


- También captura la estructura de anidamiento
- Pero se abstrae de la sintaxis concreta
  - Más compacto y fácil de usar
- Es una estructura de datos importante en un compilador

Recursive Descent Parsing

RECURSIVE DESCENT PARSING

- El árbol de análisis se construye
  - Desde la parte superior
  - De izquierda a derecha
- Los terminales se ven en orden de aparición en el flujo de tokens: ***t2 t5 t6 t8 t9***



RECURSIVE DESCENT PARSING

- Considere la gramática
  - $E \rightarrow T | T + E$
  - $T \rightarrow int | int * T | (E)$
- El flujo de tokens es: $(int5)$
- Comience con el nivel superior no terminal E
- Pruebe las reglas para E en orden

RECURSIVE DESCENT PARSING

E → T |T + E
T → int | int * T | (E)

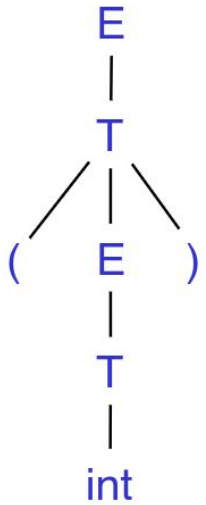
E

(int5)

RECURSIVE DESCENT PARSING

$E \rightarrow T \mid T + E$

$T \rightarrow int \mid int * T \mid (E)$



Aceptado

(int_5)

Recursive Descent Algorithm

Recursive Descent Algorithm (RDA)

Consideraciones a tomar

- TOKEN: Representa un símbolo genérico que puede ser cualquier tipo de token -e.g. INT, OPEN, CLOSE, PLUS, TIMES -

- NEXT: Es un puntero o cursor que apunta al próximo token en la secuencia de entrada que se está analizando.

Recursive Descent Algorithm (RDA)

RDA es un algoritmo que define ***funciones booleanas*** que verifiquen coincidencia de:

1) Un terminal de token dado

```
bool term(TOKEN tok) return *next++ == tok;
```

verifica si el token al que apunta NEXT coincide con el token esperado (tok)

Recursive Descent Algorithm (RDA)

2) La enésima producción de S

   `bool Sn()` ...

3) Comprueba todas las producciones de S:

   `bool S()` ...

Recursive Descent Algorithm (RDA)

Considerando la gramática:

```
E → T

E → T + E

T → int

T → int * T

T → ( E )
```

Recursive Descent Algorithm (RDA)

Para la producción E → T

```
bool E1() { return T(); }
```

Para la producción E → T + E

```
bool E2() { return T() && term(PLUS) && E(); }
```

Recursive Descent Algorithm (RDA)

Para todas las producciones de E (con backtracking)

```
bool E() {
TOKEN *save = next; //guarda el valor actual del puntero
return (next = save, E1())|| (next = save, E2()); }
```

Recursive Descent Algorithm (RDA)

Funciones para un no terminal T

```
bool T1() { return term(INT); }

bool T2() { return term(INT) && term(TIMES) && T(); }

bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() {

    TOKEN *save = next;

    return (next = save, T1()) || (next = save, T2()) ||
    (next = save, T3());

}
```

Recursive Descent Algorithm (RDA)

Para iniciar el analizador

- Inicializar al lado del punto del primer token
  - Invocar E()

- Fácil de implementar a mano

Recursive Descent Algorithm (RDA)

```
 1   bool term(TOKEN tok) { return *next++ == tok; }
 2
 3   bool E1() { return T(); }
 4   bool E2() { return T() && term(PLUS) && E(); }
 5
 6   bool E() {TOKEN *save = next; return (next = save, E 1())
 7                                      || (next = save, E2()); }
 8
 9   bool T1() { return term(INT); }
10   bool T2() { return term(INT) && term(TIMES) && T(); }
11   bool T3() { return term(OPEN) && E() && term(CLOSE); }
12
13   bool T() { TOKEN *save = next; return (next = save, T1())
14                                      || (next = save, T2())
15                                      || (next = save, T3()); }
```

Recursive Descent Algorithm (RDA)

$E \rightarrow E'|E' + id$
$E' \rightarrow -E'|id|(E)$

- ☐ Línea 3
- ☐ Línea 5
- ☐ Línea 6
- ☐ Línea 12

```
1  bool term(TOKEN tok) { return *next++ == tok; }

2  bool E1() { return E'(); }
3  bool E2() { return E'() && term(PLUS) && term(ID); }
4  bool E() {
5      TOKEN *save = next;
6      return (next = save, E1()) && (next = save,  E2());
7   }

8  bool E'1() { return term(MINUS) && E'(); }
9  bool E'2() { return term(ID); }
10 bool E'3() { return term(OPEN) && E() && term(CLOSE); }
11 bool E'() {
12     TOKEN *next = save;  return (next = save, T1())
13                                 || (next = save,  T2())
14                                 || (next = save,  T3());
15 }
```

Recursión por la izquierda

Recursión por la izquierda

- Considere una producción ***S → Sa***

```
bool S1() { return S() && term(a); }

bool S() { return S1(); }
```

- S() entra en un ciclo infinito
- Una gramática recursiva por la izquierda tiene una S no terminal ***S →+ Sα*** para alguna α
- El descenso recursivo ***no funciona*** en tales casos

Recursión por la izquierda

- Considere la gramática recursiva por la izquierda

  $S \rightarrow S\alpha \mid \beta$

- S genera todas las cadenas que comienzan con β y siguen cualquier número de α

- Se puede reescribir usando la recursividad derecha

  $S \rightarrow \beta S0$

  $S0 \rightarrow \alpha S0 \mid \varepsilon$

Recursión por la izquierda

En general:

$$S \rightarrow S\alpha_1 \mid .. \mid S\alpha_n \mid \beta_1 \mid .. \mid \beta_m$$

Todas las cadenas derivadas de S comienzan con uno de β_1,.., β_m y continúan con varias instancias de α_1,.., α_n

Reescribir como

$$S \rightarrow \beta_1 S' \mid .. \mid \beta_m S'$$

$$S' \rightarrow \alpha_1 S' \mid .. \mid \alpha_n S' \mid \varepsilon$$

Recursión por la izquierda

Considere la siguiente gramática:

```
E → E + T
  | T

T → INT
```

lo que puede llevar a un bucle infinito al intentar analizar una expresión como 1 + 2 + 3

Recursión por la izquierda

La gramática:

$S \rightarrow A\ \alpha \mid \delta$

$A \rightarrow S\ \beta$

También es recursiva por la izquierda, dado que si reemplazamos la producción A en S tendremos:

$S \rightarrow^{+} S\ \beta\ \alpha$

Recursión por la izquierda

Acerca del Descenso Recursivo

- Estrategia de análisis simple y general
  - La recursividad a la izquierda debe eliminarse primero
  - ... pero eso se puede hacer automáticamente
- Históricamente impopular debido al ***backtracking.***
  - Se pensaba que era demasiado ineficiente.
  - En la práctica, con algunos ajustes, es rápida y simple en máquinas modernas.
  - El ***backtracking*** puede ser controlado restringiendo la gramática.

Recursión por la izquierda