

Reconocimiento Descendente

Top-Down Parsing



Marco Hernández V.

Error Handling

Top-Down Parsing...

- Objetivo de los compiladores:
 - Detectar programas no válidos
 - Traducir los programas válidos
- Existe distintos tipos de errores (ej. Usando C)

Tipo de Error	Ejemplo	Detectedo por	Descripción
Léxico	... \$...	Lexer	Usa caracteres no reconocidos
Sintaxis	...X*%....	Parser	No hace sentido y no se puede compilar
Semántico	...int x; y=x(3);...	Type Checker	No existe correspondencia
Correctitud	Cualquier programa	Tester, usuario	No se puede realizar lo esperado

Top-Down Parsing...

- El manejador de errores (Error handler) debe:
 - Reportar los errores de una manera exacta y clara
 - Recuperarse de un error de forma rápida
 - No perjudicar la velocidad en la compilación de código válido
- Hay tres tipos de manejador de errores:
 - Modo pánico
 - Producción de errores
 - Corrección automática local o global

Top-Down Parsing...

- En el Pasado
 - Ciclo de recompilación lenta (incluso una vez al día)
 - Encontrar tantos errores como sea posible en un ciclo
- En el presente
 - Ciclo de recompilación rápido
 - Usuarios tienden a corregir el error
 - Recuperación de errores complejos es menos irresistible

Árbol de Sintaxis abstracta

Top-Down Parsing...

Árbol de Sintaxis

- Un parseador determina la derivación de una secuencia de tokens, pero el resto del compilador necesita una representación estructural del programa.
- Los árboles de sintaxis se parecen a los árboles de parseo pero ignoran algunos detalles.
- Abreviado como AST

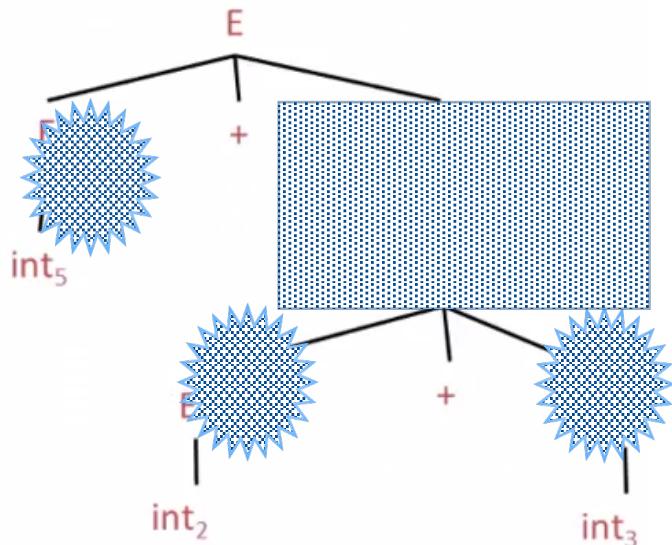
Top-Down Parsing...

Árbol de Sintaxis

- Considere la gramática $E \rightarrow \text{int} \mid (E) \mid E + E$

- Y la hilera $5 + (2 + 3)$

- Un análisis léxico (lista de tokens) $\text{int}_5 ' + ' (' \text{int}_2 ' + ' \text{int}_3 ')'$

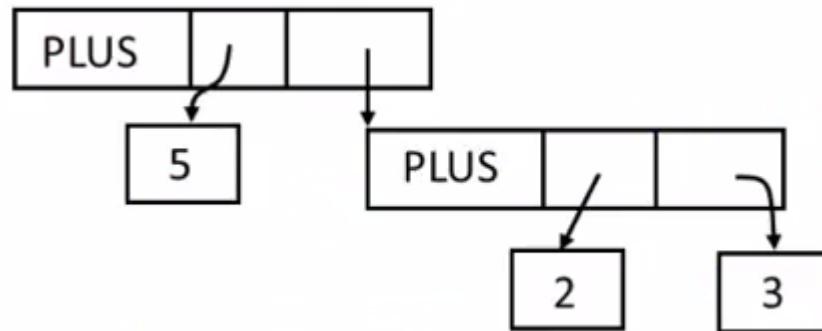


Este árbol de parser:

- Determina la operación del parser
- Captura estructura anidada
- Demasiada información
 - Parentesis
 - Nodos de sucesión simple

Top-Down Parsing...

Árbol de Sintaxis



- Captura la estructura anidada
- Pero lo abstrae de la sintaxis concreta
 - Más compacto y fácil de utilizar
- Una importante estructura de datos en un compilador

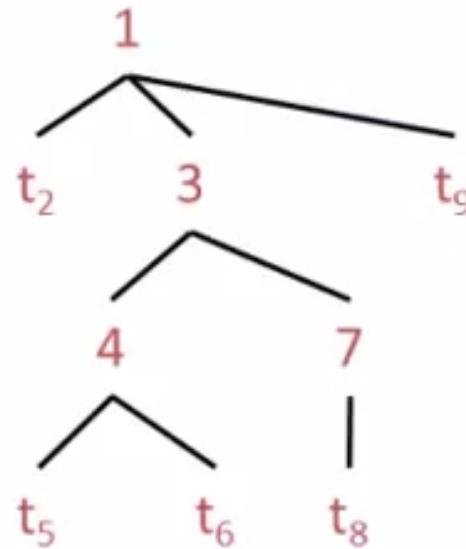
Algoritmo de Parseo Recursive Descent Parsing

Top Down Parsing

Top-Down Parsing...

- El árbol de parseo es construido:
 - Desde arriba
 - De izquierda a derecha
- Los terminales son vistos en orden de aparición en el flujo de tokens

$t_2 \ t_5 \ t_6 \ t_8 \ t_9$



Top-Down Parsing...

- Considere la gramática

$$E \rightarrow T \mid T + E$$

$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

- Con la siguiente hilera de tokens:
 (int_5)
- Se aplicará el parseo utilizando la estrategia “recursive descendant” iniciando con el No terminal E
 - Top Down desde la raíz
 - De Izquierda a derecha
 - Probar las producción en orden
 - Cuando la producción falla se “echa” para atrás para probar otras producciones

Top-Down Parsing...

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

(int₅)
↑

E
|
T
|
int

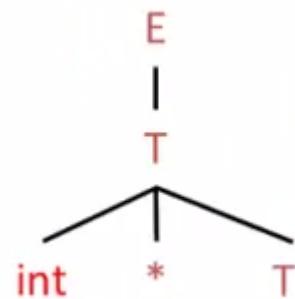


Top-Down Parsing...

$$E \rightarrow T \mid T + E$$

$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

(int₅)
↑



No es el camino
correcto !!!!

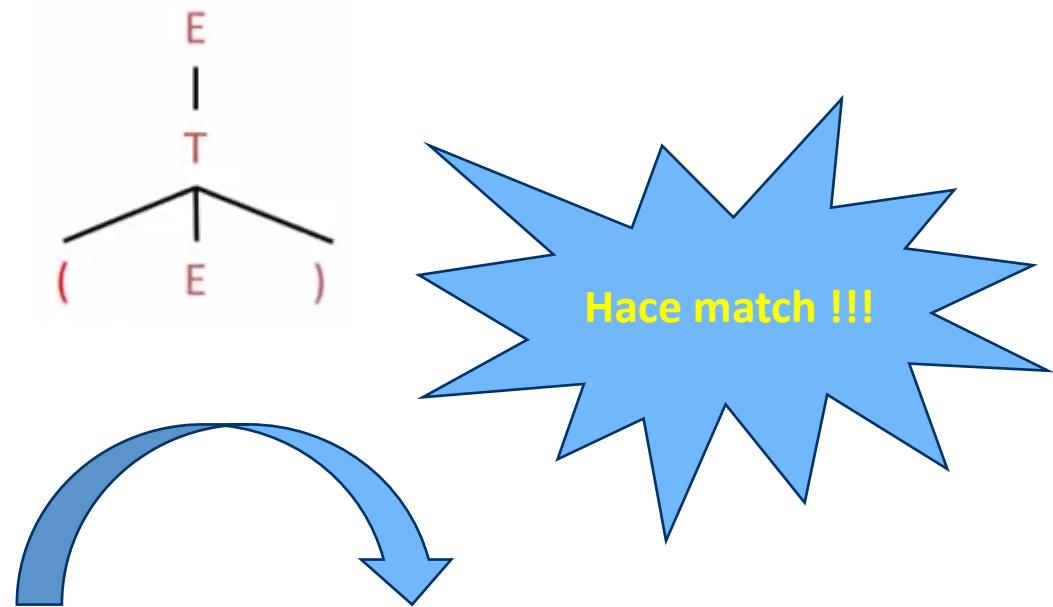


Top-Down Parsing...

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

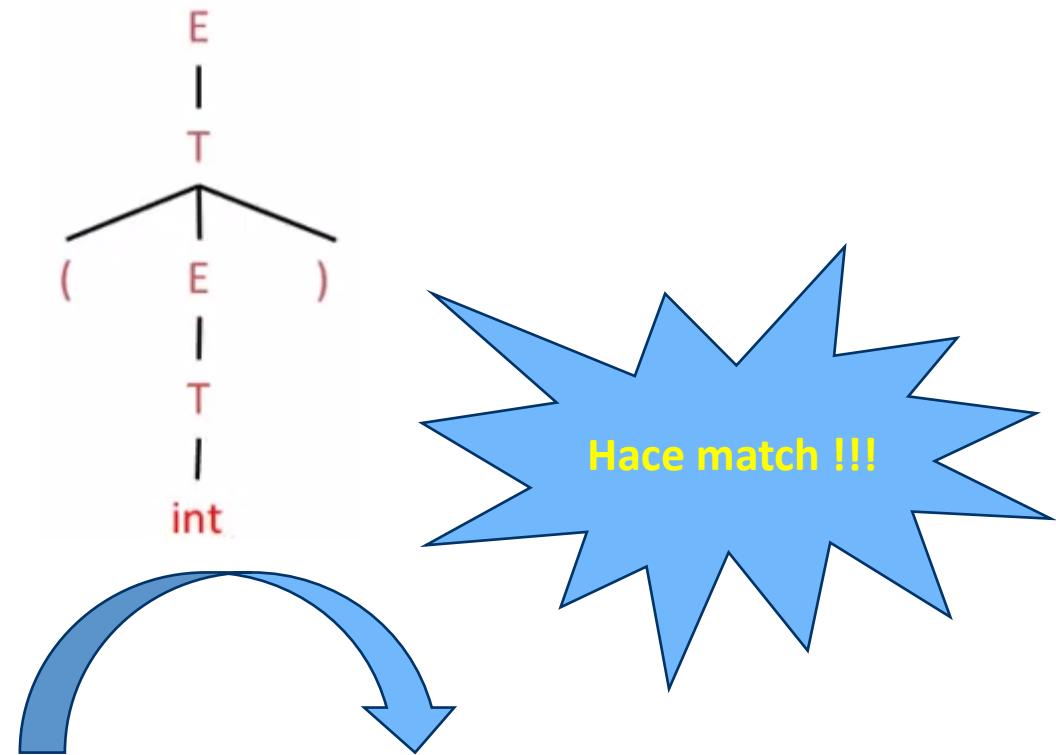
(int₅)
↑↑



Top-Down Parsing...

$$E \rightarrow T \mid T + E$$
$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

(int₅)
↑↑↑

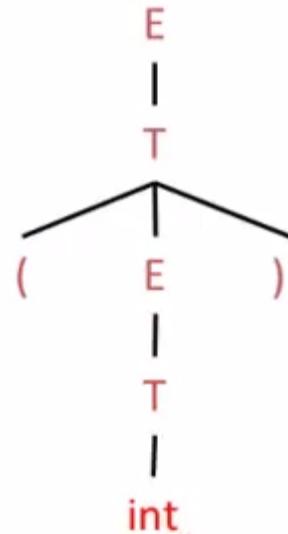


Top-Down Parsing...

$$E \rightarrow T \mid T + E$$

$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

(int₅)
↑↑↑



Ejercicios

- Dada la siguiente gramática:

$$\begin{aligned} E &\rightarrow T + E \\ E &\rightarrow T \\ T &\rightarrow F * T \\ T &\rightarrow F \\ F &\rightarrow a \\ F &\rightarrow b \\ F &\rightarrow (E) \end{aligned}$$

- Reconocer la entrada: $(a + b) * a + b$
- Utilizando reconocimiento descendente con retroceso.

Algoritmo

Top-Down Parsing...

- Se definen una serie de funciones booleanas
 - Un token terminal dado

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

Dado el token (enviado por parámetro, se verifica si hace match con la actual posición de la entrada, de tal manera que si es “verdadero” entonces se apunta a la siguiente entrada.

- La producción nth de S

bool $S_n()$ { ... }

Solamente valida el éxito de una producción de s

- Prueba todas las producciones de S

bool $S()$ { ... }

Prueba todas las producciones de S

Top-Down Parsing...

$$\begin{aligned} E &\rightarrow T \mid T + E \\ T &\rightarrow \text{int} \mid \text{int} * T \mid (E) \end{aligned}$$

- Para la producción $E \rightarrow T$

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

- Para la producción $E \rightarrow T + E$

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

- Para todas las producciones de E

```
bool E() {
    TOKEN *save = next;
    return (next = save, E1())
        || (next = save, E2());
}
```

Top-Down Parsing...

$$\begin{aligned} E &\rightarrow T \mid T + E \\ T &\rightarrow \text{int} \mid \text{int} * T \mid (E) \end{aligned}$$

- Funciones para la 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()); }
```

Top-Down Parsing...

$$\begin{aligned} E &\rightarrow T \mid T + E \\ T &\rightarrow \text{int} \mid \text{int} * T \mid (E) \end{aligned}$$

- Para iniciar el parseo
 - Inicialice “NEXT” en la ubicación del primer token e invoke $E()$

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

2

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

3

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

1

```
bool E() { TOKEN *save = next; return (next = save, E1())  
          || (next = save, E2()); }
```

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

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

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

4

```
bool T() { TOKEN *save = next; return (next = save, T1())  
          || (next = save, T2())  
          || (next = save, T3()); }
```

Limitaciones

Top-Down Parsing...

$$E \rightarrow T \mid T + E$$
$$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$$

int
INT

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

2

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

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

1

```
bool E() { TOKEN *save = next; return (next = save, E1())
```



```
                || (next = save, E2()); }
```

4

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



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



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

3

```
bool T() { TOKEN *save = next; return (next = save, T1())
```



```
                || (next = save, T2());
```



```
                || (next = save, T3()); }
```



Top-Down Parsing...

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

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

int * int



2
bool $E_1()$ { return $T()$; }
bool $E_2()$ { return $T()$ && term(PLUS) && $E()$; }

Se elimina la entrada !!!

1
bool $E()$ {TOKEN *save = next; return (next = save, $E_1()$)
 || (next = save, $E_2()$); }

4
bool $T_1()$ { return term(INT); }
bool $T_2()$ { return term(INT) && term(TIMES) && $T()$; }
bool $T_3()$ { return term(OPEN) && $E()$ && term(CLOSE); }

3
bool $T()$ { TOKEN *save = next; return (next = save, $T_1()$)
 || (next = save, $T_2()$)
 || (next = save, $T_3()$); }

Top-Down Parsing...

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

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

2 bool E₁() { return T(); }

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

1 bool E() { TOKEN *save = next; return (next = save, E₁())
 || (next = save, E₂()); }

4 bool T₁() { return term(INT); }
 bool T₂() { return term(INT) && term(TIMES) && T(); }
 bool T₃() { return term(OPEN) && E() && term(CLOSE); }

3 bool T() { TOKEN *save = next; return (next = save, T₁())
 || (next = save, T₂())
 || (next = save, T₃()); }

int * int



No es el camino
correcto

¿qué paso?

Cuando se encontró que en T se encontró el éxito, no hubo la oportunidad de “devolverse” y buscar otra posible solución para T !!!

Top-Down Parsing...

$$\begin{aligned} E &\rightarrow T \mid T + E \\ T &\rightarrow \text{int} \mid \text{int} * T \mid (E) \end{aligned}$$

int * int

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

2 bool E₁() { return T(); }

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

1 bool E() { TOKEN *save = next; return (next = save, E₁())
 || (next = save, E₂()); }

4 bool T₁() { return term(INT); }
 bool T₂() { return term(INT) && term(TIMES) && T(); }
 bool T₃() { return term(OPEN) && E() && term(CLOSE); }

3 bool T() { TOKEN *save = next; return (next = save, T₁())
 || (next = save, T₂())
 || (next = save, T₃()); }



Si se hubiese descubierto
esto... nuestro lenguaje
hubiese sido exitoso !!!

El problema es el “retroceso” dentro de una producción...

Top-Down Parsing...

$$\begin{aligned} E &\rightarrow T \mid T + E \\ T &\rightarrow \text{int} \mid \text{int} * T \mid (E) \end{aligned}$$

int * int

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

2 bool E₁() { return T(); }

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

1 bool E() { TOKEN *save = next; return (next = save, E₁())
 || (next = save, E₂()); }

4 bool T₁() { return term(INT); }
 bool T₂() { return term(INT) && term(TIMES) && T(); }
 bool T₃() { return term(OPEN) && E() && term(CLOSE); }

3 bool T() { TOKEN *save = next; return (next = save, T₁())
 || (next = save, T₂())
 || (next = save, T₃()); }

No es el camino correcto
¿qué paso?

No hay “backtracking” una vez que se ha encontrado una producción que satisface para un No Terminal !!! No hay forma de buscar una mejor solución !!!

Top-Down Parsing...

- Si una producción es satisfactorio para un No Terminal X
 - No se puede retroceder para intentar una distinta producción para X
- El algoritmo “Recursive Descendent ” es muy genérico y funciona para muchas clases de gramática principalmente donde para cualquier No Terminal a lo máximo hay una producción que puede tener éxito.
- La gramática ejemplo podría ser reescrita para trabajarla con este algoritmo
 - Usando *left factoring* (se verá más adelante)

Left Recursion

Top-Down Parsing...

- Considere la producción

$$S \rightarrow S a$$

bool $S_1()$ { return $S()$ && term(a); }

bool $S()$ { return $S_1()$; }

- $S()$ podría generar un loop infinito.

- Una gramática left recursive tiene un NoTerminal S

$$S \rightarrow^+ S\alpha \text{ para algún } \alpha$$

- Recursive Descendent no trabaja correctamente en tales casos.

Top-Down Parsing...

- Considere la siguiente gramática left-recursive

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

- S genera todas las hileras de inicio con un β y seguido por cualquier número de α 's

- Podría generar:

$$S \rightarrow S\alpha \rightarrow S\alpha\alpha \rightarrow S\alpha\alpha\alpha \rightarrow S\alpha\ldots\alpha \rightarrow \beta\alpha\ldots\alpha$$

- Por esto no sirve en Recursive Descendent Parsing debido a que se requiere ver siempre la primer parte de la entrada y entonces trabajar de izquierda a derecha

Top-Down Parsing...

- Se puede reescribir usando right-recursión

$$\begin{aligned} S &\rightarrow \beta S' \\ S' &\rightarrow \alpha S' \mid \epsilon \end{aligned}$$

- Y puede producir

$$S \rightarrow \beta S' \rightarrow \beta \alpha S' \rightarrow \beta \alpha \alpha S' \rightarrow \beta \alpha \dots \alpha S' \rightarrow \beta \alpha \dots \alpha$$

- Genera la misma hilera que la gramática anterior.

Top-Down Parsing...

- En Resumen, dado

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

- Todas las hileras derivadas de S inician con β_1, \dots, β_m y continúan con varias instancias de $\alpha_1, \dots, \alpha_n$

- Pueden ser reescritas

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

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

Top-Down Parsing...

- La gramática

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

$$A \rightarrow S\beta$$

$$S \rightarrow A\alpha \rightarrow S\beta\alpha$$

- Es además left-recursive debido a

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

- Presenta la misma situación, y se puede solucionar de la misma manera.

Top-Down Parsing...

- Recursive Descendent
 - Estrategia de parseo sencilla y general
 - Left-recursión debe ser eliminada
 - Se puede eliminar de forma automática pero generalmente se realiza de forma manual (cambiando la gramática)

Reconocimiento Descendente

Top-Down Parsing

