Tecnológico de Costa Rica

Compiladores e Intérpretes

Grupo 40

Profesor: Dr. Francisco Torres Rojas

Apuntes de clase: 05 de mayo de 2017

Apuntador: Víctor Andrés Chaves Díaz

Carné: 2015107095

2017

Tabla de contenidos

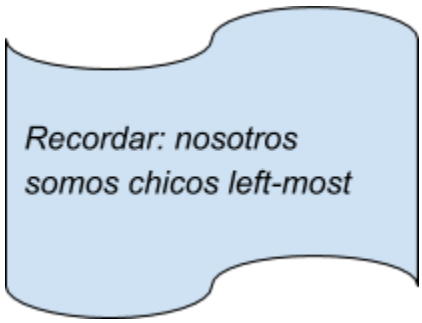
Ambigüedad - Continuación **2**
- Ambigüedad en CFG 2
- Precedencia 3
- CFG y Precedencia 3
- Asociatividad 7
- Asociatividad y CFG 8

Análisis Sintáctico - Parsing **10**
- Parse! - El juego del parsing 10
- ¿Cómo jugar al Parse!? 11

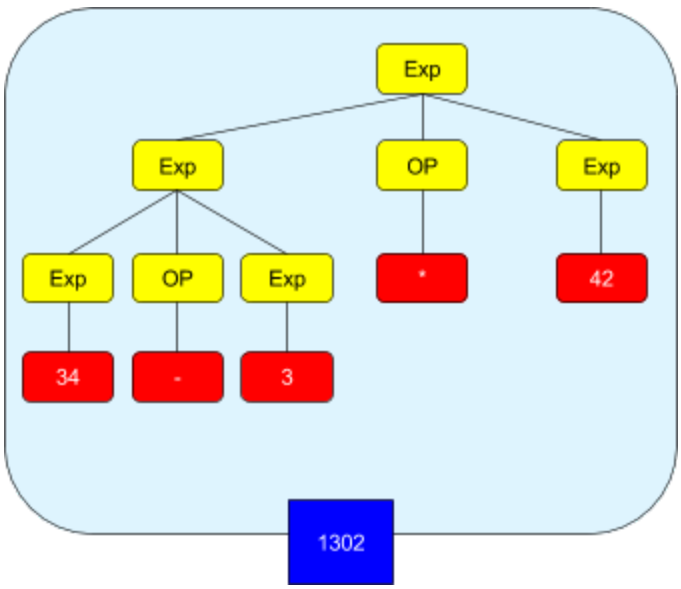
Análisis Sintáctico - Parsing Predictivo **12**
- Predicciones 12
- Elementos del Parsing Predictivo 12
- Tabla de acciones 12
- Algoritmo de Parsing Predictivo 13
- Acciones 13
- Ejemplos 14
  - Ejemplo 1 14
  - Ejemplo 2 15
  - Ejemplo 3 16
  - Ejemplo 4 17

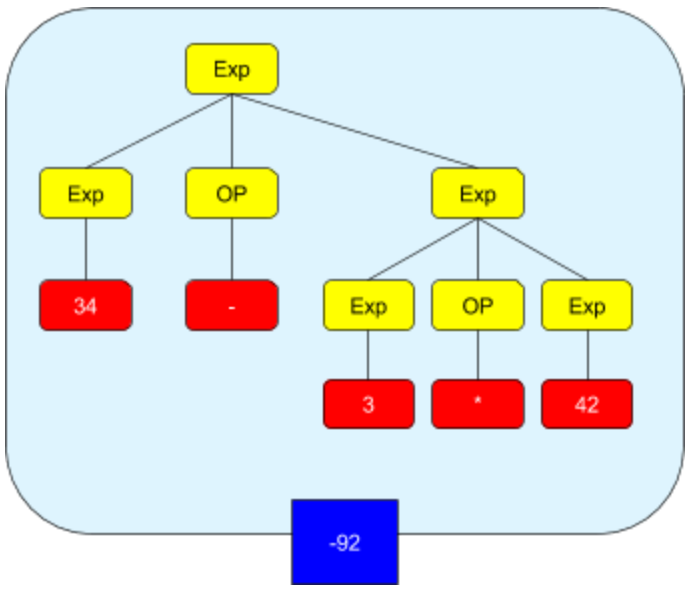
Ambigüedad - Continuación

Se inicia la clase haciendo un repaso del último tema visto el miércoles: ambigüedad.



También volvemos a repasar el problema de los dos árboles que genera la hilera 34 - 3 * 42





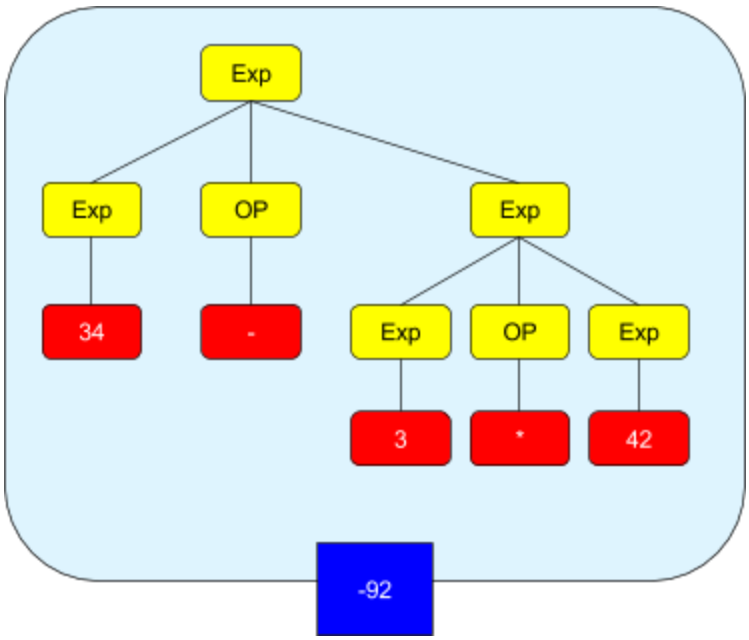
Ambigüedad en CFG

- ¿Cómo se demuestra que una CFG es ambigua? Simplemente se elige una hilera que pueda generar más de un árbol diferente.
- Y el contrario, ¿cómo demuestro que mi CFG *no* es ambigua? Desafortunadamente, no existe un solo algoritmo conocido para resolver problemas de ambigüedad.
- Si sabemos que una CFG es ambigua, ¿que podemos hacer? Crear reglas de desambiguación.
- Los lenguajes de programación pueden tener ambigüedades, pero por regla general tienen mecanismos para manejar dichos problemas.

Precedencia

- Se crea una precedencia entre dos operadores o construcciones, esto indica cual de los dos se ejecuta antes que el otro (y debe verse reflejado en la gramática).
- Estas reglas de precedencia son puramente arbitrarias.
    - Se dice (cuento del profe) que hace mucho tiempo se hizo una reunión entre los más grandes matemáticos, así como el Concilio de Nicea, para discutir sobre la precedencia de las operaciones.
- Pueden existir parejas que no tengan una precedencia establecida.
- Se decide si se lee de izquierda a derecha o su versión contraria.
- Ejemplos clásicos:
    - Multiplicación/división sobre suma/resta.
    - Exponenciación sobre multiplicación.
    - Paréntesis sobre todos estos.

Un ejemplo visto previamente, `34 - 3 * 42`, utilizando las convenciones previamente mencionadas, ahora solo nos puede generar un árbol, el correcto según nuestra precedencia:



CFG y Precedencia

Ahora que sabemos que podemos forzar precedencia en nuestras gramáticas, la pregunta que queda es: ¿cómo diablos se refleja la precedencia en una CFG?

Existe un truco muy sencillo para saber esto, una sencilla frase: *Los últimos serán los primeros*.

Al usted escuchar esta frase, probablemente su expresión es esta:

A lo que me refiero con dicha frase es que, como ya sabemos, la gramática se puede expresar como un árbol, y resulta que la precedencia se refleja en este árbol al estar más lejano del símbolo inicial.

Podemos ver que:

- Las reglas que se encuentren más cercanas al símbolo de inicio tienen menor precedencia, se utilizarán primero y por tanto quedarán más arriba en el árbol de derivación
- Por el contrario, las reglas más alejadas al símbolo de inicio tienen mayor precedencia, se utilizarán de último y por tanto quedarán más abajo en el árbol de derivación.

Esto se vuelve un juego de machetear la gramática y acomodarla de manera que se refleje esta jerarquía. Para demostrar esto, tomemos la gramática:

```
Exp → Exp OP Exp
Exp → (Exp)
Exp → <número>
OP  → + | - | *
```

Y hagamos los cambios necesarios para que la multiplicación preceda a la suma y resta,

Agarrense el sombrero...
Redoble de tambores…

```
Exp    → Exp ADDOP Exp | Term
ADDOP  → + | -
```

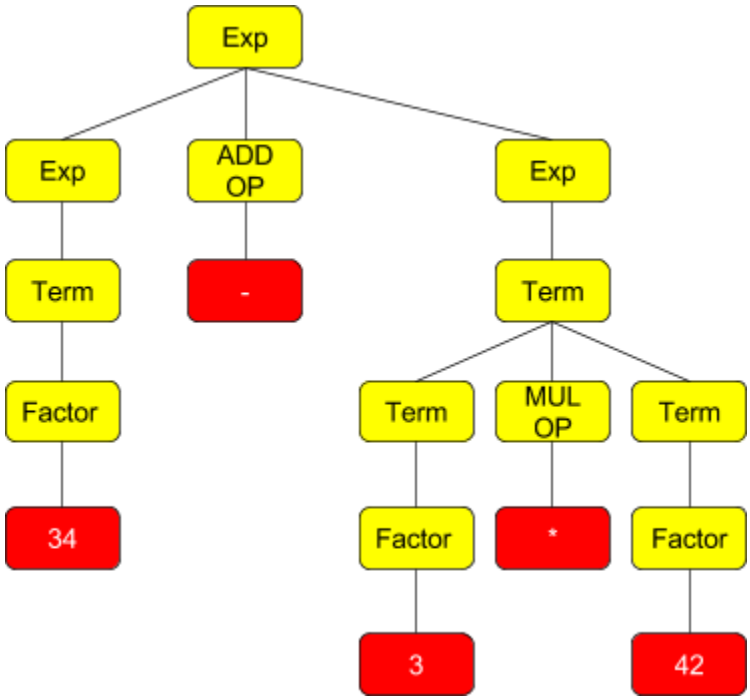
```
Term   → Term MULOP Term | Factor
MULOP  → *
Factor → (Exp) | <número>
```

Es valido que si no recuerda (o no estaba presente en clase) estos cambios, su expresión sea la siguiente:

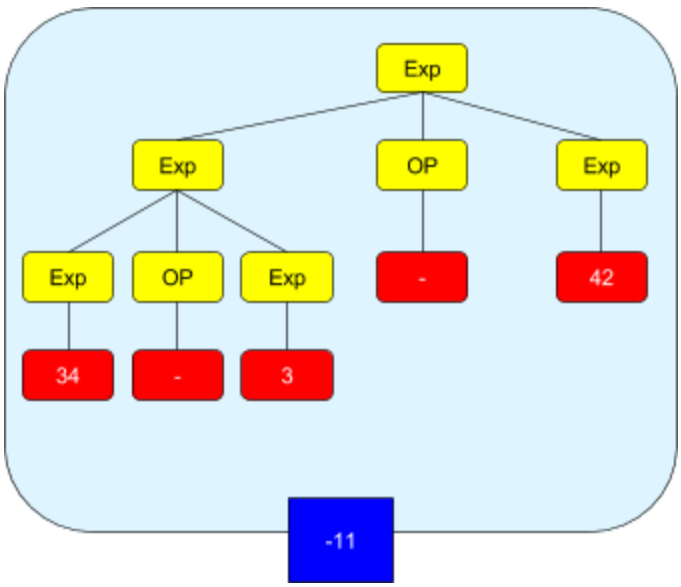
(Ya nos hacía falta el mae de Mind = Blown)

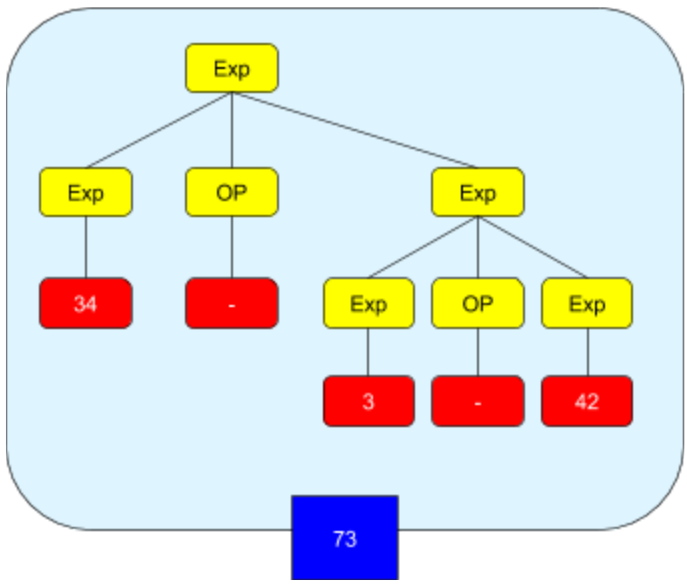
Para explicar los cambios que se han hecho, la pieza más importante que se introduce en esta nueva gramática es `Term`, una regla que puede ser usada por `Exp` y que es el único que puede crear una multiplicación (`MULOP`) o un `Factor`. Esta regla es la que crea que la multiplicación se encuentre en un nivel inferior que la suma en el árbol de derivación.

Interesantemente, ambas gramáticas mostradas son equivalentes, pues generan las mismas hileras, la diferencia reside en el hecho de que la segunda genera menos árboles de derivación (exactamente lo que buscamos). Para poder verificar esto, se puede intentar todo lo que quiera generar, con la hilera `34 - 3 * 42`, el árbol izquierdo (al inicio de los apuntes están los dos árboles) con la nueva gramática y será imposible, el único que se puede generar es:



No obstante, nuestra gramática sigue incompleta, pues hileras como `34 - 3 - 42` siguen generando dos árboles diferentes.

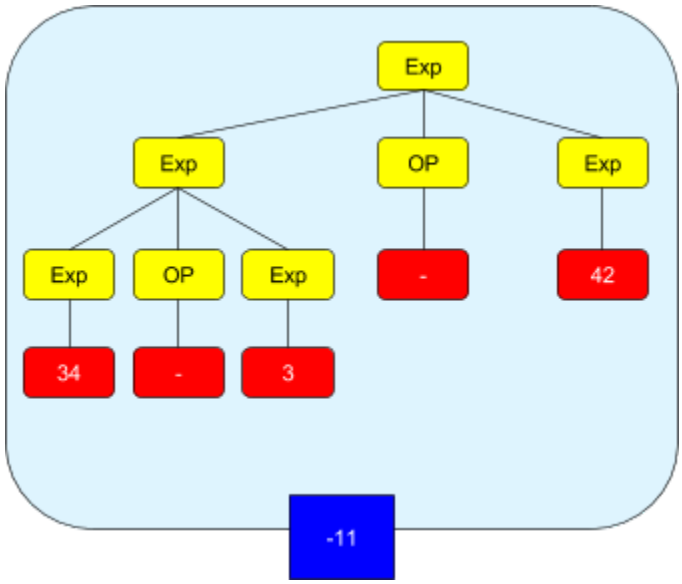




Asociatividad

- En el caso de que se tengan dos operadores o construcciones del mismo tipo (es decir, que ninguno preceda al otro), es necesario que se establezca el tipo de asociatividad entre dichos operadores/construcciones.
- Puede ser por la izquierda o por la derecha.
- Uno de los dos se debe ejecutar antes que el otro.
- La gramática debe reflejar esta asociatividad.
- Ejemplos clásicos:
    - Multiplicación, división, suma y resta son asociativas por la izquierda.
    - Asignación en C es asociativa por la derecha.
    - *Puntero a* en C es asociativo por la derecha.
- Por lo general, las referencias de los lenguajes de programación tienen una tabla especificando la asociatividad de los operadores.

Un ejemplo visto previamente, `34 - 3 - 42`, utilizando la convención de suma y resta previamente mencionadas, ahora solo nos puede generar un árbol, el correcto según nuestra asociatividad:



Asociatividad y CFG

Podemos ver que el profesor hizo auto-plagio de una diapositiva anterior, ya que tiene el mismo error de "estable" en vez de "establece"

- Similar a la precedencia, se utiliza la altura del operador en el árbol de derivación para indicar la asociatividad.
- Asumiendo que tomamos un operador X, el cual es asociativo por la izquierda, y viene dos veces:
    - El operador X a izquierda extrema queda en un subárbol más bajo en el árbol de derivación.
    - El operador X a derecha extrema queda en un subárbol más arriba en el árbol de derivación.

Para ejemplificar esto, podemos mostrarlo modificando la gramática para reflejar esta asociatividad.

Previamente se tenía:

```
Exp    → Exp ADDOP Exp | Term
ADDOP  → + | -
Term   → Term MULOP Term | Factor
MULOP  → *
Factor → (Exp) | <número>
```

Y vuelvan a agarrarse del sombrero, porque aquí viene la nueva gramática: Redoble de tambores…

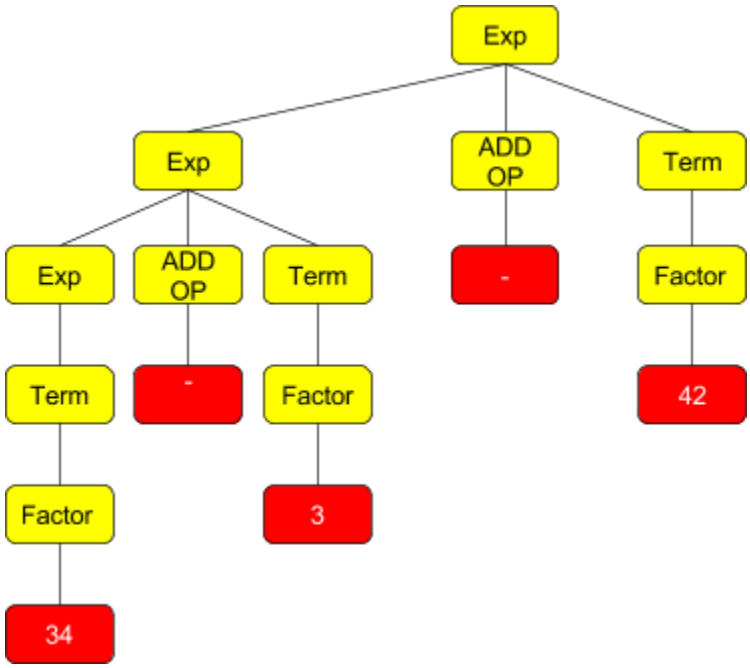
```
Exp    → Exp ADDOP Term | Term
ADDOP  → + | -
Term   → Term MULOP Factor |
Factor
MULOP  → *
Factor → (Exp) | <número>
```

A Kennedy también le parecía interesante la resolución de asociatividad

"Pero, entre esta y la otra gramática, ¿cual es la diferencia?". Tal vez con un poco de colores las diferencias sean más notables:

```
Exp    → Exp ADDOP Term | Term
ADDOP  → + | -
Term   → Term MULOP Factor | Factor
MULOP  → *
Factor → (Exp) | <número>
```

En efecto, con esos dos cambios se resuelve el dilema de asociatividad. Lo que ocurre, utilizando el ejemplo de `34 - 3 - 42`, es que para poder hacer restas, se debe expandir `Exp`, y solo se puede expandir por el lado *izquierdo* de la regla, algo igual pasa con la multiplicación, como se puede apreciar en el siguiente árbol:



En caso de que quisiera cambiar y utilizar asociatividad por la derecha, cambio las parejas de `Exp/Term` y `Term/Factor`.

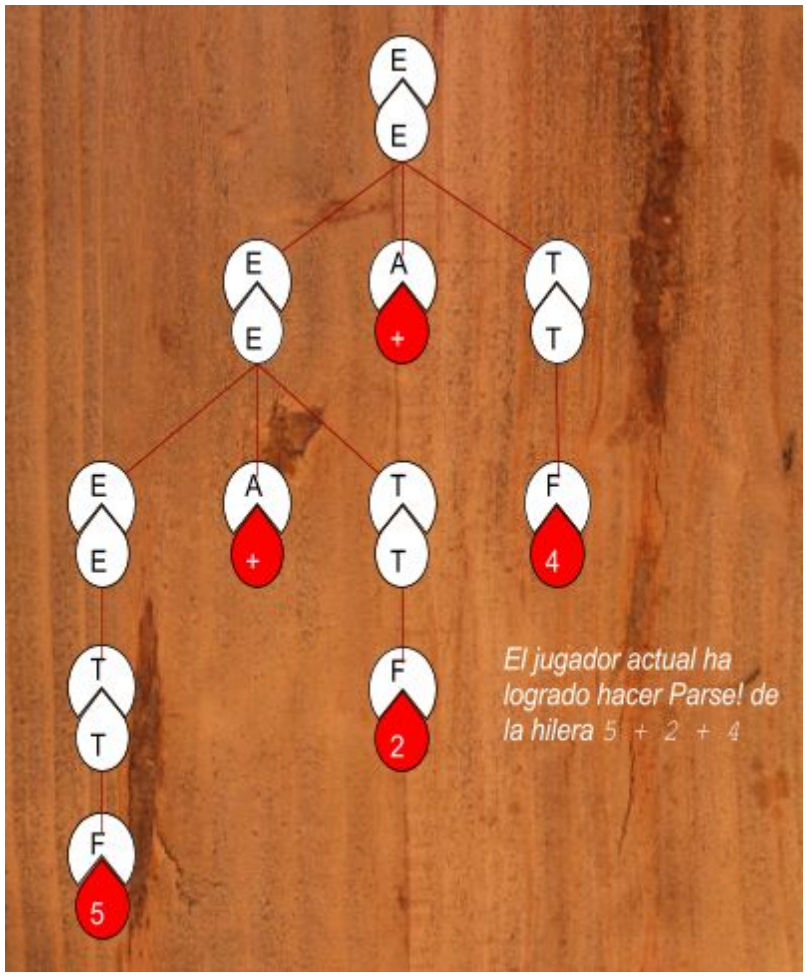
Para finalizar este tema, por regla general (heurísticas), si se tiene el mismo no terminal dos veces en una regla (como en `Exp → Exp ADDOP Exp | Term`), es muy probable que ocasione problemas de asociatividad.

Análisis Sintáctico - Parsing

- Una CFG *genera* un CFL (nos podemos reír de la gente que diga que los reconoce).
- ¿Y que usamos para *reconocer* CFLs? ¡Pues un parser!
- El parser es una máquina binaria, se le ingresa una hilera y devuelve si es parte del lenguaje libre de contexto o no
    - Como un efecto secundario, genera un árbol de derivación si la hilera es parte del lenguaje, ya sea explicita o implícitamente.

Parse! **- El juego del parsing**

2 - 4 jugadores
Edades 7+
Precio: $199.99



El parsing, como visto en clase, se puede ver como un tipo de juego, así como el cricket:

- Al inicio del juego, se saca una CFG.
- Existe una bolsa mágica que genera infinitas piezas intermedias (fichas similares a las del Scrabble) que corresponden a diferentes reglas de la gramática elegida.
- En cada partida, se nos dá una hilera (ya tokenizada) que debemos parsear con la gramática.
- En la parte superior de la mesa, colocamos el símbolo inicial de la CFG.
- Utilizando las piezas correctas, se debe conectar de manera perfecta el símbolo inicial con todos los tokens que conforman la hilera de entrada.
- Cuando falte poner una sola pieza, el jugador debe gritar *Parse!* (no realmente).
- Si se puede lograr conectar el símbolo inicial con todos los tokens, la hilera pertenece al lenguaje.
- En caso de que NO se pueda, se debe indicar que existe un error sintáctico.

Una regla muy importante del parsing:

NO SE PUEDE HACER BACKTRACKING!

… Esto nos deja en una posición desventajosa, ¿cómo se hace para saber cual pieza colocar?

Muy sencillo: con un poco de magia que se verá a continuación.

¿Cómo jugar al *Parse!*?

- La primera pregunta que se debe hacer: ¿en qué dirección nos movemos?
  - Si nos movemos desde el símbolo inicial hacia la hilera, nos movemos de manera *top down*, como hacía nuestro compilador de MICRO.
  - En cambio, si nos movemos desde la hilera hacia el símbolo inicial, nos movemos de manera *bottom up*, como lo hace GNU Bison.
- La segunda pregunta: ¿qué pieza pongo?
  - Obviamente, la correcta :v
- … ¿y como hacemos para saber cual pieza usar?
  - Como somos chicos left-most, podemos ver cual es el no terminal abierto a izquierda derecha.
  - Adicionalmente, podemos ver el siguiente símbolo de la hilera (esto se llama un *lookahead*).

- En el caso de nosotros, vamos a aprender a jugar usando *top down* y *left-most*.
- Y, ¿podemos decidir con precisión cuál regla usar?

Análisis Sintáctico - Parsing Predictivo

Es importante recalcar: si el parser se equivoca una sola vez, está *malo* y hay que tirarlo a la calle.

Este compilador está malo -- Walter Saborío, compañero de trabajo del profe

Predicciones

- Sabemos que hay que elegir en cada paso la regla a utilizar.
- Cómo usamos leftmost derivation, sabemos cual es el no terminal actual a izquierda extrema. Aún así, hay muchas posibles derivaciones de esta regla.
- Podemos ver los primeros *k* tokens de la hilera (un lookahead de *k* tokens).
- Con esto, podemos utilizar una técnica llamada ***parsing predictivo***, que como su nombre lo indica, utilizando solamente los *k* tokens de lookahead y el no terminal actual, podemos saber con exactitud la regla a utilizar, o si hay que reportar un error de sintaxis.
- Y, ¿qué pasa si usamos k = 1?

Elementos del Parsing Predictivo

Para poder hacer parsing predictivo, se requiere de 4 componentes importantes:
IMAGEN

- La gramática libre de contexto.
- La hilera (con un lookahead de 1).
- Una pila.
- Una tabla de acciones.
    - Cae del cielo como maná (no la banda).

Estudiantes de Compiladores e Intérpretes recolectando tablas de acciones dadas por su Señor Torres, 2017

Tabla de acciones

Como su nombre lo indica es una tabla (esa nadie se la esperaba), que está compuesta por los siguientes datos:

- Las filas se componen de la mezcla entre los símbolos terminales y no terminales.
- Las columnas solo contienen los símbolos terminales.
- Para poder sub-indexar la fila, se utiliza el símbolo en el tope de la pila.
- Para poder sub-indexar la columna, se utiliza el lookahead actual.

- Utilizando la fila y la columna, llegamos a una casilla específica, la cual contiene… ¡la acción a ejecutar!
- Afortunadamente para nosotros, existen algoritmos que automatizan la creación de la tabla de acciones.

Algoritmo de Parsing Predictivo

- Se debe agregar un símbolo más a tanto la tabla como al final de la hilera de entrada, el símbolo $.
- Se hace un push en la pila, de $.
- Se hace un push en la pila, del símbolo inicial de la gramática.
- Se establece el lookahead inicial.
- Se hace un ciclo:
    - Mientras la hilera no se rechace o acepte, ejecute la acción en `tabla_acciones[top_pila][lookahead]`.

Acciones

- Aceptar:
    - Se ubica en la intersección de $ y $.
    - Si se llega a este estado, se detiene la ejecución del parseo y se devuelve que la hilera es aceptada por la gramática.
    - Se acepta la hilera completa.
    - Indica que la hilera si es derivable usando la CFG.
- Rechazar:
    - Equivale a un error.
    - Se rechaza la hilera completa.
    - Indica que la combinación del top de la fila y el lookahead es inaceptable a lo que le concierne la gramática.
    - Puede ser una entrada vacía.
    - Implica que la hilera no es derivable usando la CFG.
- Match:
    - El top de la pila es un terminal.
    - El top es igual al lookahead actual
    - Efectos:
        - Se avanza en la hilera de entrada (pasa el lookahead al siguiente token).
        - Se hace un pop de la pila.
    - No es necesarios hacerlos explícitos en la tabla, se puede hacer match implícito.

- Expandir:
    - El top de la pila contiene a un no terminal.

- La casilla de acción contiene una regla, o el número de regla que se debe ejecutar.
- Efectos:
  - Se hace un pop del top de la pila.
  - Se hace push de los elementos al lado derecho de la regla, de manera que el símbolo en izquierda extrema quede en el top de la pila (este símbolo puede ser ε).

Ejemplos

Ejemplo 1

Se tiene la gramática:

1. S → (S)S
2. S → ε

Y la siguiente tabla de acciones:

	(	)	$
S	1	2	2
(	match	error	error
)	error	match	error
$	error	error	accept

Y la hilera `(()())`

Pasos

```
$S              (()())$         →   Expandir con regla 1
$S)S(                (()())$        →   Match
$S)S            (()())$         →   Expandir con regla 1
$S)S)S(         (()())$         →   Match
$S)S)S          (()())$         →   Expandir con regla 2
$S)S)                (()())$        →   Match
$S)S            (()())$         →   Expandir con regla 1
$S)S)S(         (()())$         →   Match
$S)S)S          (()())$         →   Expandir con regla 2
$S)S)                (()())$        →   Match
$S)S            (()())$         →   Expandir con regla 2
$S)             (()())$         →   Match
$S              (()())$         →   Expandir con regla 2
$               (()())$         →   Aceptar :)
```

Ejemplo 2

Se tiene la gramática:

3. S → (S)S
4. S → ε

Y la siguiente tabla de acciones:

	(	)	$
S	1	2	2
(	match	error	error
)	error	match	error
$	error	error	accept

Y la hilera `()()`

Pasos

```
$S          ()()$        →   Expandir con regla 1
$S)S(            ()()$   →     Match
$S)S        ()()$        →   Expandir con regla 2
$S)         ()()$        →   Match
$S          ()()$        →   Expandir con regla 1
$S)S(            ()()$   →     Match
$S)S        ()()$        →   Expandir con regla 2
$S)         ()()$        →   Match
$S          ()()$        →   Expandir con regla 2
$           ()()$        →   Aceptar :)
```

Ejemplo 3

Se tiene la gramática:

5. S → (S)S
6. S → ε

Y la siguiente tabla de acciones:

	(	)	$
S	1	2	2
(	match	error	error
)	error	match	error
$	error	error	accept

Y la hilera `)()`

Pasos

```
$S              )()$          →    Expandir con regla 2
$               )()$          →    Error :(
```

Ejemplo 4

Se tiene la gramática:

```
1. E    → T E'
2. E'   → OP T E'
3. E'   → ε
4. Op   → +
5. Op   → -
6. T    → F T'
7. T'   → M F T'
8. M    → *
9. T'   → ε
10.  F → (E)
11.  F → #
```

Y la siguiente tabla de acciones (se obvian las filas de símbolos terminales):

	(	#	)	+	-	*	$
E	1	1					
E’			3	2	2		3
Op				4	5		
T	6	6					
T’			9	9	9	7	9
M						8	
F	10	11					

Y la hilera `3 + 4`

Pasos

$E	3+4$	→	Expandir con regla 1
$E'T	3+4$	→	Expandir con regla 6
$E'TF	3+4$	→	Expandir con regla 11
$E'T#	3+4$	→	Match
$E'T	3+4$	→	Expandir con regla 9
$E'	3+4$	→	Expandir con regla 2
$E'T Op	3+4$	→	Expandir con regla 4
$E'T +	3+4$	→	Match
$E'T	3+4$	→	Expandir con regla 6
$E'TF	3+4$	→	Expandir con regla 11
$E'T#	3+4$	→	Match
$E'T	3+4$	→	Expandir con regla 9
$E'	3+4$	→	Expandir con regla 3
$	3+4$	→	Aceptar :)