

Transparencias de Autómatas y desarrollo avanzado del software.

Máster en Ingeniería de Telecomunicación

Gorka Guardiola

GSYC

23 de noviembre de 2022



Este trabajo se entrega bajo la licencia “Atribución-CompartirIgual 4.0 Internacional” [1] (cc-by-sa).

Usted es libre de:

- ▶ **Compartir:** *Copiar y redistribuir el material en cualquier medio o formato.*
- ▶ **Adaptar:** *Remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.*
- ▶ **Se pueden dispensar estas restricciones si se obtiene el permiso de los autores.**
- ▶ *Las imágenes de terceros mantienen sus derechos originales.*

©2022 Gorka Guardiola.

[1] Algunos derechos reservados. “Atribución-CompartirIgual 4.0 Internacional” (cc-by-sa). Para obtener la licencia completa, véase <https://creativecommons.org/licenses/by-sa/4.0/deed.es>. También puede solicitarse a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Puedes conseguir la última versión de este documento en:

<https://github.com/honecomp/honecomp.github.io/raw/main/slides/comp.pdf>

- ▶ Tema 0: Introducción y Objetivos del Curso
- ▶ Tema 1: Scanner/Lexer, tokens
- ▶ Tema 2: Análisis sintáctico
- ▶ Tema 3: Parsing descendente recursivo
- ▶ Tema 3.1: Parsing descendente de expresiones
- ▶ Tema 4: Acciones y atributos
- ▶ Tema 5: Símbolos y ámbitos
- ▶ Tema 6: Parsing ascendente
- ▶ Tema 7: Yacc
- ▶ Tema 8: Tipos de datos
- ▶ Tema 9: Generación de código
- ▶ Tema 10: Implementación de un intérprete

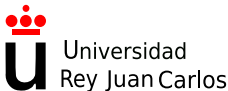
Tema 0: Introducción y Objetivos del Curso

Autómatas y desarrollo avanzado de software

Gorka Guardiola Múzquiz

GSYC

23 de noviembre de 2022



¿Cual es el programa que más ejecuta un programador?

- El compilador o intérprete (C, Python, Java, JS, Go...)
- Hay que estar familiarizado, entender diferencias, errores, bugs...
 - ▶ ¿Qué significa lvalue?
 - ▶ ¿Qué compilador de C elijo?

¿Cual es el programa que más ejecuta un desarrollador de hardware digital?

- El compilador o intérprete
- VHDL, Verilog, Spice. . .
- El software se come el mundo, pero fundamentalmente, ¿qué es?
- Lenguajes de programación, de descripción. . .

Lenguajes formales

- Útiles en general
- Límites de máquina de estados, autómatas, parsing
- Expresiones regulares, globbing, configuración. . .
- Minilenguajes, describir patrones formales
- Potente: Turing, Church, límites de la computación, las matemáticas como lenguaje

¿Qué es un compilador/intérprete?

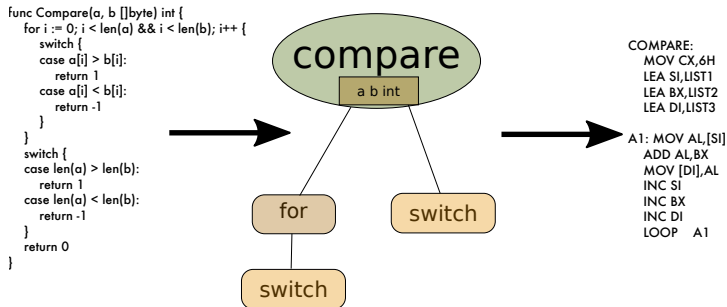
- Un traductor de un lenguaje formal, de programación, a otro, código máquina (Ej: C a x86_64)
- Un intérprete traduce a una máquina abstracta, ejecuta
- Buen libro de referencia el *“libro del dragón”*

Aho, Lam, Sethi, Ullman, Compilers: Principles, Techniques, and Tools, Pearson International Edition, 2006. ISBN 0-321-49169-6

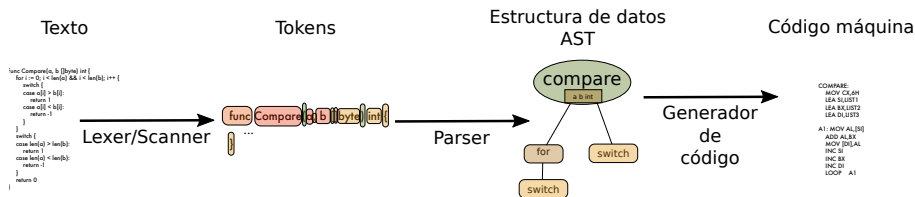
¿Qué es un compilador?

- Traducen a código máquina
- Construye una estructura de datos en memoria (árbol/grafó) y luego genera código
- Pueden traducir a otro lenguaje (transpilador o traductor), (Ej: C a Java)
- Puede generar estructuras de datos (Ej: generador de stubs)

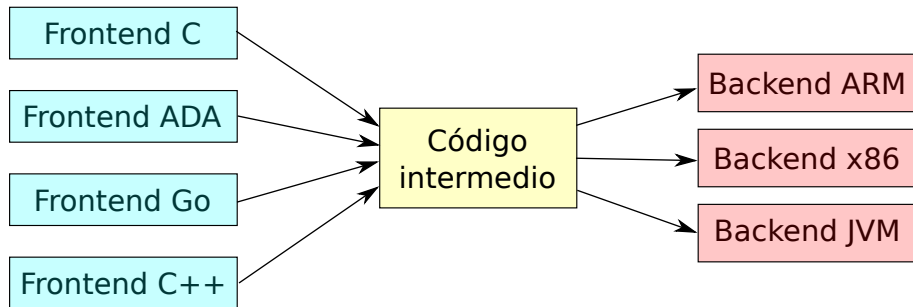
¿Qué es un compilador?



¿Qué es un compilador?

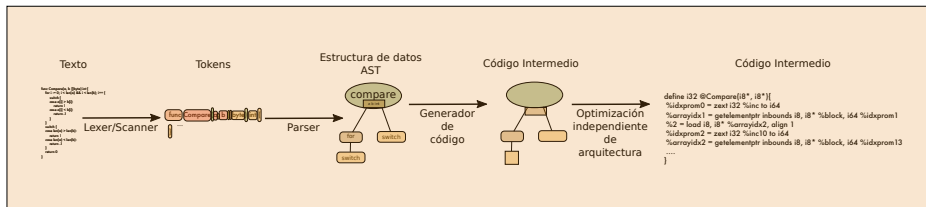


¿Qué es un compilador?

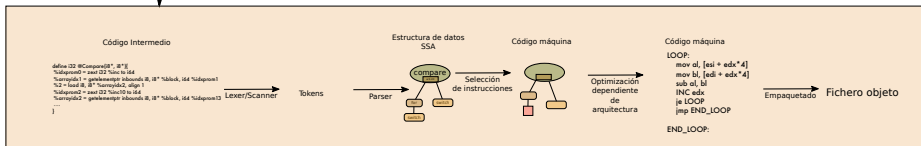


¿Qué es un compilador?

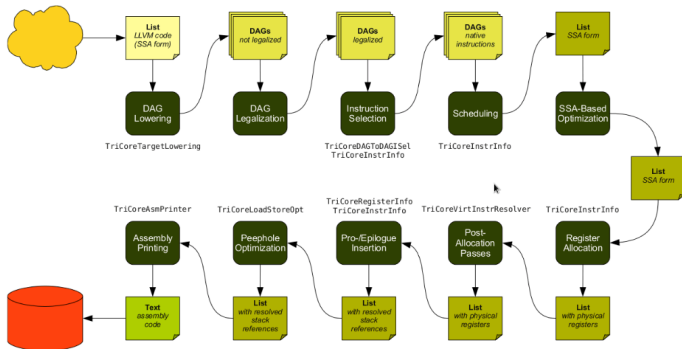
FRONTEND



BACKEND



Ejemplo backend llvm



¿Qué es un intérprete?

- Procesa menos el código, lo tiene que hacer en tiempo real
- Crea una estructura de datos en memoria como antes y luego la ejecuta
- El código fuente está “ejecutando”

¿Qué es un lenguaje formal?

- Abstraer el concepto de lenguaje
- Tengo un conjunto de símbolos válidos y una serie de tuplas válidas (o más cómodo, descripciones de tuplas)
- Hay formas de describir qué es válido (enumeración, gramáticas, BNF...)
- Al final tengo un algoritmo que ejecuta y que dice si una tupla es válida o no, es parte del lenguaje o no (y luego se puede recuperar la estructura)
- La tupla vacía puede ser parte del lenguaje

Lenguajes formales en programación: ejemplo

- Describir los programas *válidos*
- Ver un programa
 - ▶ En tiempo de compilación: un compilador
 - ▶ En tiempo de ejecución: un intérprete
- Encontrar errores y decírselo al usuario
- No sólo (sirven para muchas más cosas)

Ejemplo lenguaje formal

- Por enumeración (lenguaje finito)
https://en.wikipedia.org/wiki/Formal_language
- Lenguaje es $\Sigma = \{a, b\}$ lenguaje es $L = \{\epsilon, a, ab, ba, bb\}$
- Represento la cadena vacía con ϵ
- Si el lenguaje es gordo, es terrible (puede ser infinito...).

Ejemplo lenguaje formal

- Usando expresiones regulares, (puede ser infinito)
https://en.wikipedia.org/wiki/Regular_expression
- Lenguaje es $\Sigma = \{a, b\}$ lenguaje es $L = a^*b$, que es
 $L = \{b, ab, aab, aaab, aaaab, \dots\}$
- Tengo los operadores de repetición de Kleene, $*$, $+$ y el opcional $?$
- Paréntesis para agrupar (precedencia de operadores) $()$
- Operador alternativa $|$, dos expresiones a y b $a|b$ significa que la expresión puede encajar con el primero o el segundo
- Operador escape $\backslash$ para que un carácter pierda su significado especial, por ejemplo $\backslash()$ es el terminal paréntesis
- Operador concatenación (no se escribe, como la multiplicación), sencillamente juxtapuesto
- Conjuntos, cualquier terminal, el punto: $.$ y conjunto, los corchetes:
 $[a - c]$
- Delimitadores de anclado como $^$, $\$$ (principio y fin)

Expresiones regulares: equivalencias

- Algunos operadores y construcciones son redundantes
- $[a - c] \equiv a|b|c$
- $x* \equiv x+$

Expresiones regulares: precedencia

Arriba, más precedencia, por ejemplo $ab* \equiv a(b*)$ la concatenación tiene menos prioridad que el asterisco.

- 1 Clases de equivalencia (no las hemos visto: *collation*)
- 2 Escape \
- 3 Conjuntos definidos entre corchetes $[a - c]$
- 4 Agrupaciones entre paréntesis (*hola*)
- 5 Operadores de Kleene, $*$, $+$ y $?$
- 6 Concatenación (por yuxtaposición)
- 7 Delimitadores de anclado $\$, ^$
- 8 Alternancia $a|b$

Expresiones regulares

- Son un metalenguaje (igual que las matemáticas son un metalenguaje)
- Es un lenguaje formal (expresiones regulares) para describir lenguajes formales (patrones de texto)

Ejemplo lenguaje formal

- Usando gramáticas formales generativas
https://en.wikipedia.org/wiki/Formal_grammar
- Tengo reglas de reescritura (reglas de producción) P , símbolos terminales Σ y no terminales N
- Las tuplas del lenguaje están formadas por símbolos terminales (el alfabeto del lenguaje)

Ejemplo lenguaje formal

- Lenguaje es L reconocido por la
- gramática G con símbolos terminales $\Sigma = \{a, b\}$
- símbolos no terminales $N = \{S, B\}$
- de los cuales S es el símbolo inicial
- reglas de producción
 - 1 $S \rightarrow aB$
 - 2 $aB \rightarrow bSb$
 - 3 $B \rightarrow aa$

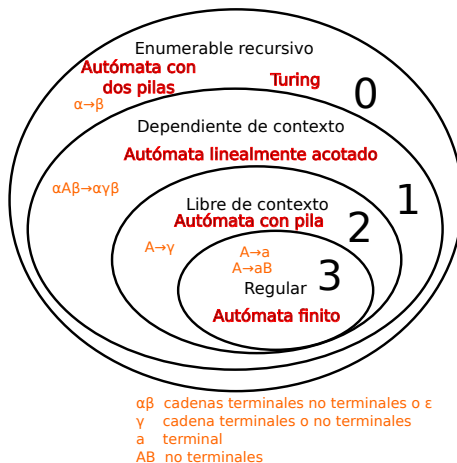
Ejemplo lenguaje formal

- Empiezo con S , aplico $S \rightarrow aB$, tengo aB
- Aplico $aB \rightarrow bSb$, tengo bSb
- Aplico de nuevo $S \rightarrow aB$ y tengo $baBb$
- Aplico $B \rightarrow aa$ y tengo $baaab$
- A cualquier cadena que pueda llegar aplicando las reglas que quiera hasta no tener símbolos no terminales
- A eso se le llama **derivación**

Ejemplo lenguaje formal

- Los lenguajes puede ser más complejos (expresivos) o más simples (menos expresivos)
- El tipo de gramática que se puede usar para describirlos, hay una jerarquía
 - ▶ La jerarquía es por subconjuntos. Para conjuntos infinitos, recordar que pueden tener igual cardinalidad (ej: los pares y los enteros).
- Luego hay clasificaciones que no encajan limpiamente (veremos más adelante)
- También el tipo de máquina/algorithm que se puede usar para reconocerlos

Jerarquía de Chomsky



Chomsky, Noam (1956). "Three models for the description of language". IRE Transactions on Information Theory (2): 113–124.

Jerarquía de Chomsky

- Lenguajes **regulares**: tokens, máquinas de estados, expresiones regulares
- Lenguajes **libres de contexto**: lenguajes de programación, XML, (Dyck: paréntesis)
- Lenguajes **dependientes de contexto**: Declaraciones en ámbitos
- Enumerable **recursivo**: las matemáticas, todos

Regular

- Tipo-3
- $A \rightarrow a$ y $A \rightarrow aB$
- a terminal, AB no terminales
- Autómata finito (finite-state machine FSM or finite-state automaton FSA)

Regular

- Expresiones regulares
- Tokenización
- Máquinas de estados

Cómo programar un reconocedor de expresiones regulares usando máquinas de estado:

<https://swtch.com/~rsc/regexp/regexp1.html> y <https://swtch.com/~rsc/regexp/> .

Autómata finito, FSM

- Máquina de estados
- Dos estados especiales, inicio y fin
- Puede haber varios finales, también conocidos como aceptadores
- Cada entrada avanza a otro estado

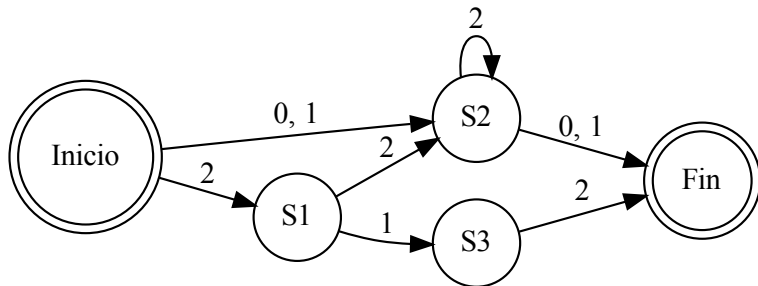
Autómata finito, FSM (DFA)

- Lenguaje de entrada, terminales $\Sigma = \{a, b, c\}$
- Conjunto de estados Q
- Función de transición $\delta : Q \times \Sigma \rightarrow Q$ (otra forma de verlo es como tripletas estado origen, entrada, destino)
- Estado inicial *inicio* $\in Q$
- Estados aceptadores $F \subseteq Q$
- Estado actual $q_a \in Q$

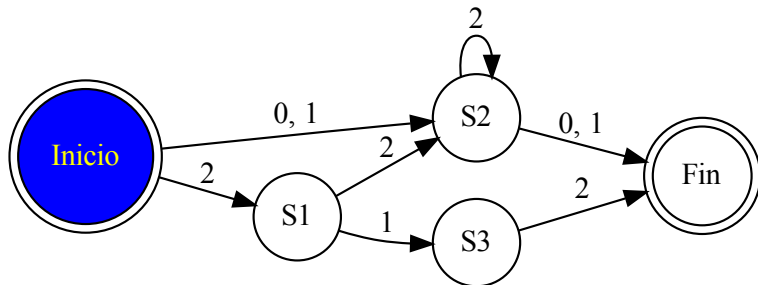
FSM de Moore y Mealy

- Vamos a estudiar **DFA/NDFA** (FSM) no es exactamente ni de Moore ni de Mealy
- **Mealy**: la salida depende entrada y estado actual
- **Moore**: la salida depende del estado actual sólo
- Ambas: más importantes para electrónica (pero matemáticamente equivalentes)
- Puede haber entradas vacías (veremos que no encajan bien en esta clasificación)
- Suelen dar salida sólo en el estado aceptador, aunque a veces más similar a Moore (o a Mealy)
- Hay variaciones en los formalismos. ¿Qué nos interesa capturar con nuestra abstracción?

Autómata finito, FSM (DFA)

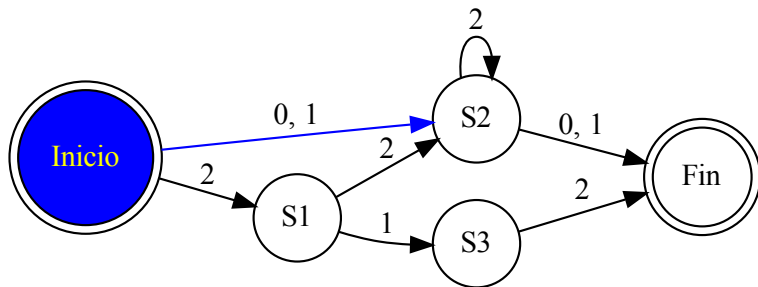


Autómata finito, FSM (DFA)



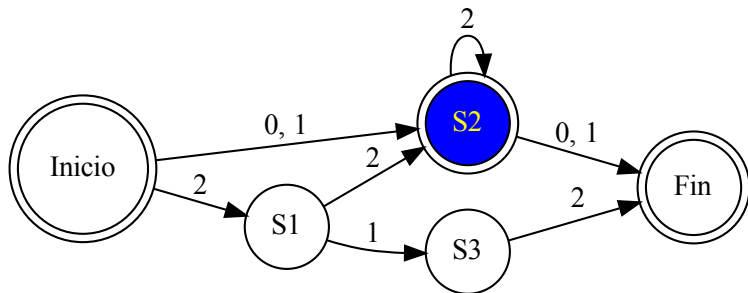
Entrada 021

Autómata finito, FSM (DFA)



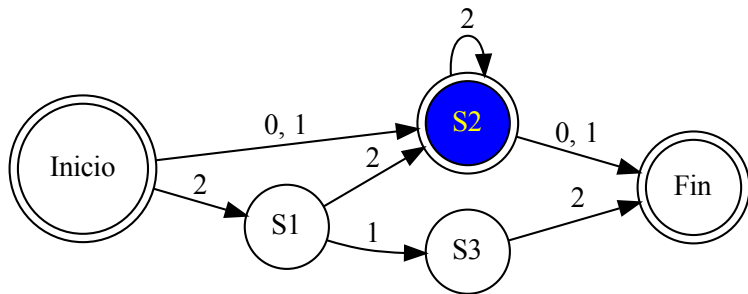
Entrada **0**₂₁

Autómata finito, FSM (DFA)



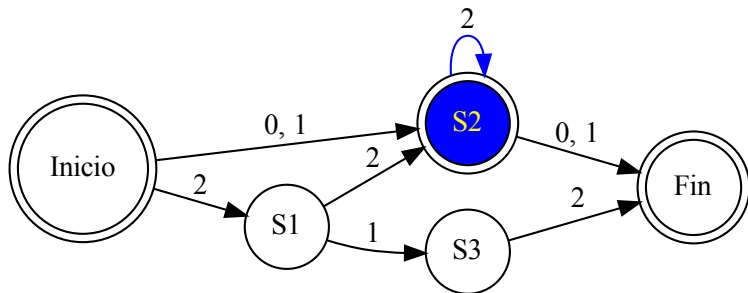
Entrada **0**₂₁

Autómata finito, FSM (DFA)



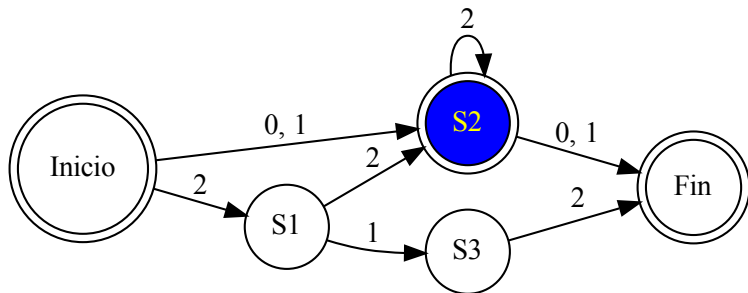
Entrada **2**
01

Autómata finito, FSM (DFA)



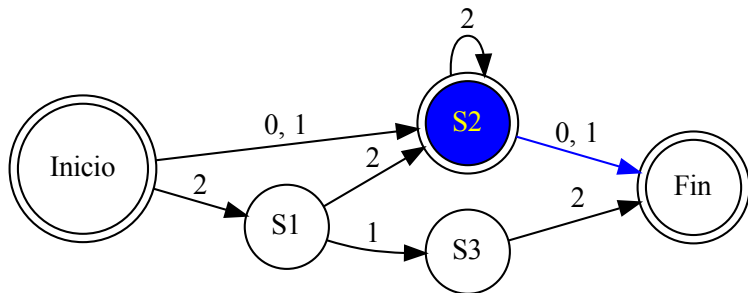
Entrada 0**2**1

Autómata finito, FSM (DFA)



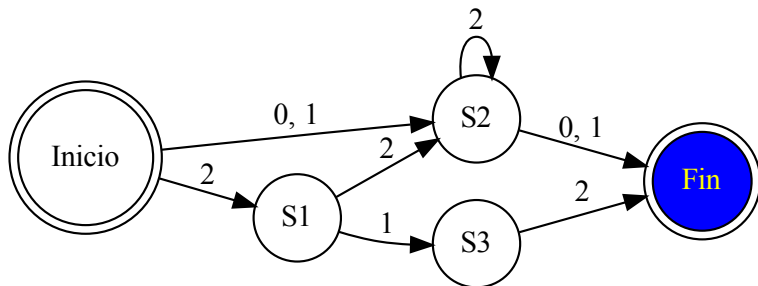
Entrada 0**2**1

Autómata finito, FSM (DFA)



Entrada 02**1**

Autómata finito, FSM (DFA)



Entrada 02**1**✓

Autómata finito, FSM

- Dos tipos, determinista (DFA) y no-determinista
- El determinista el que hemos visto

Autómata finito, NDFA/NFA

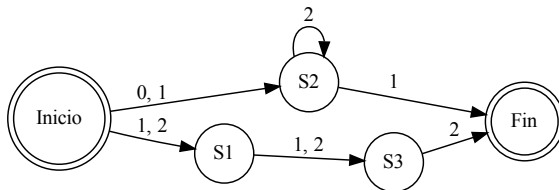
- Ojo, se llama así pero **no es aleatorio ni estocástico**
- Cada entrada puede avanzar a varios estados
- En lugar de tener un estado actual hay varios
- Tiene transiciones vacías

Rabin, M. O.; Scott, D. (1959). "Finite automata and their decision problems". IBM Journal of Research and Development. 3 (2): 114–125. doi:10.1147/rd.32.0114. ISSN 0018-8646. <http://www.cse.chalmers.se/~coquand/AUTOMATA/rs.pdf>

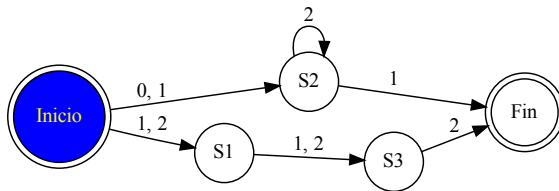
Autómata finito, FSM (NDFA)

- Lenguaje de entrada, terminales $\Sigma = \{a, b, c\}$
- Conjunto de estados Q
- Función de transición $\delta : Q \times R \rightarrow Q$, donde R es un conjunto de conjuntos de elementos de Σ , incluyendo el conjunto vacío
- Estado inicial $inicio \in Q$
- Estados aceptadores $F \subseteq Q$
- Estados actuales $Q_a \subseteq Q$

Autómata finito, NDFA

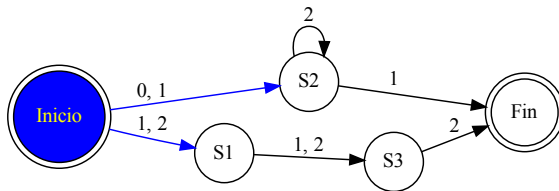


Autómata finito, NDFA



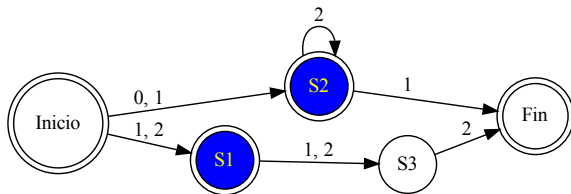
Entrada 122

Autómata finito, NDFA



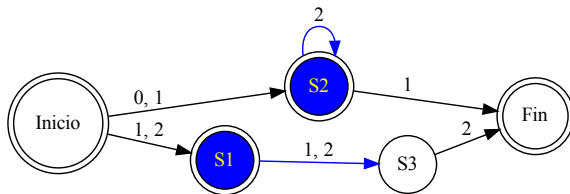
Entrada **1**₂₂

Autómata finito, NDFA



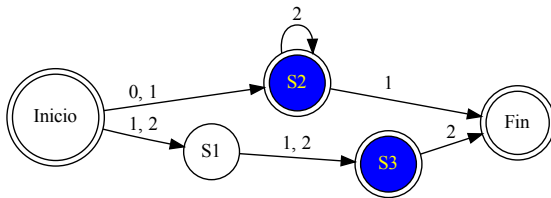
Entrada **1**₂₂

Autómata finito, NDFA



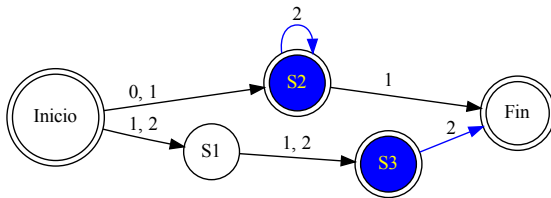
Entrada 1**2**2

Autómata finito, NDFA



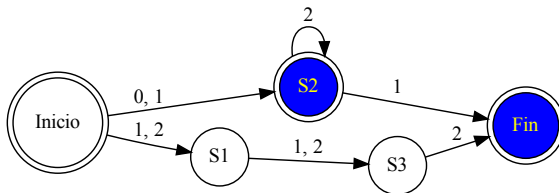
Entrada 1**2**2

Autómata finito, NDFA



Entrada 12**2**

Autómata finito, NDFA



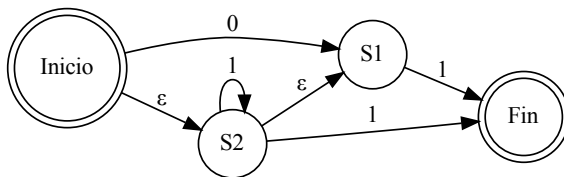
Entrada 12**2**✓

NDFA en DFA

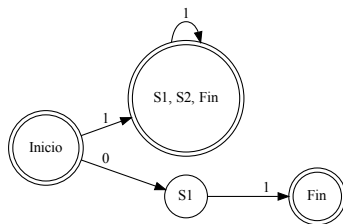
- Ambos son equivalentes
- Construcción de conjunto potencia de Rabin-Scott
- Se empieza por todos los estados a los que se puede llegar con ϵ
- Estados nuevos que son clases de equivalencia
- Todos los estados a los que se puede llegar con igual entrada
- Un NDFA de n estados puede llegar a ser un DFA de 2^n estados
- Puede ser impráctico si n es grande, depende del autómata concreto

Rabin, M. O.; Scott, D. (1959). "Finite automata and their decision problems". IBM Journal of Research and Development. 3 (2): 114–125. doi:10.1147/rd.32.0114. ISSN 0018-8646. <http://www.cse.chalmers.se/~coquand/AUTOMATA/rs.pdf>

NDFA en DFA



NDFA en DFA

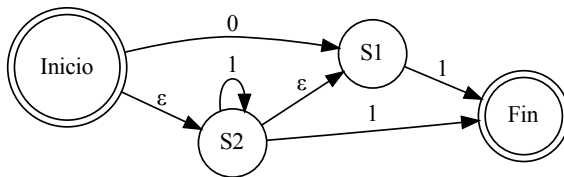


FSM (DFA o NDFA) en una tabla

- Un autómatata se puede describir con una tabla (la función de transición)

FSM en una tabla

Actual	Entrada	Siguiente
Inicio	0	S1
Inicio	ϵ	S2
S1	1	Fin
S2	1	S2
S2	ϵ	S1
S2	1	Fin



Minimización DFA

- Generado automáticamente (expresión regular, generador de código, etc.)
- Dos pasos, eliminar estados inaccesibles y eliminar estados equivalentes
 - ▶ Eliminación de estados inaccesibles: desde la entrada hacer todas las transiciones y colorear, borrar el resto (antes y después del siguiente paso)
 - ▶ Eliminar estados equivalentes: algoritmo de Hopcroft, (hay otros, Moore, etc.)

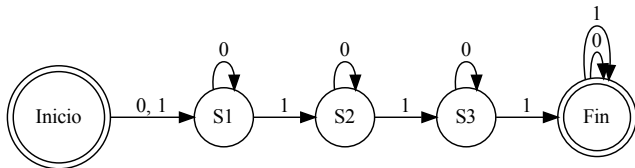
Hopcroft, John (1971), "An $n \log n$ algorithm for minimizing states in a finite automaton", Theory of machines and computations (Proc. Internat. Sympos., Technion, Haifa, 1971), New York: Academic Press, pp. 189–196, MR 0403320. See also preliminary version, Technical Report STAN-CS-71-190, Stanford University, Computer Science Department, January 1971.

Minimización DFA: Hopcroft

- Dividir estados en clases de equivalencia
- Ir partiéndolas hasta que sólo queden los no equivalentes
- Se usa la tabla de la función de transición

Minimización DFA: Hopcroft

Actual	Entrada	Siguiente	Salida
Inicio	0	S1	0
Inicio	1	S1	0
S1	0	S1	0
S1	1	S2	0
S2	0	S2	0
S2	1	S3	0
S3	0	S3	0
S3	1	Fin	0
Fin	0	Fin	1
Fin	1	Fin	1

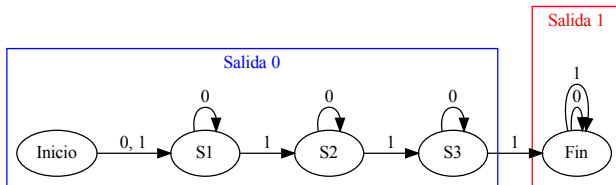


Minimización DFA: Hopcroft

- El primer paso es particionar en clases de equivalencia por salidas

Minimización DFA: Hopcroft

Actual	Entrada	Siguiente	Salida
Inicio	0	S1	0
Inicio	1	S1	0
S1	0	S1	0
S1	1	S2	0
S2	0	S2	0
S2	1	S3	0
S3	0	S3	0
S3	1	Fin	0
Fin	0	Fin	1
Fin	1	Fin	1

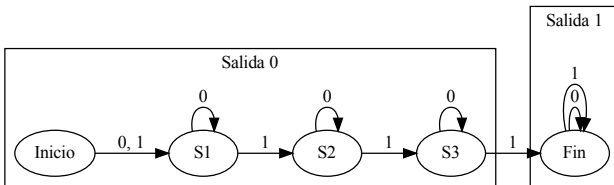


Minimización DFA: Hopcroft

- El segundo paso es invertir la tabla (para conseguir todos los predecesores)

Minimización DFA: Hopcroft

Actual	Entrada	Anterior	Salida
Inicio	0	-	0
Inicio	1	-	0
S1	0	Inicio, S1	0
S1	1	Inicio	0
S2	0	S2	0
S2	1	S1	0
S3	0	S3	0
S3	1	S2	0
Fin	0	Fin	1
Fin	1	S3, Fin	1

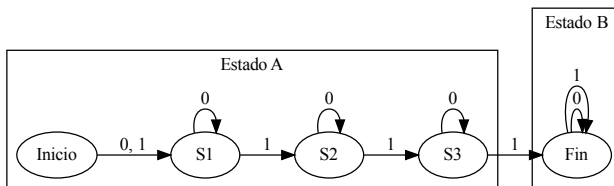


Minimización DFA: Hopcroft

- Se va uno a uno por todas las clases de equivalencia de la partición, se meten en una lista de trabajo L
- Se extrae un estado de trabajo q de la lista L de nuevos estados.
- Para cada uno de los estados de la lista Q de todos los estados se miran los símbolos.
- Cada estado equivalente de la lista L se parte en estado en 2
 - ▶ Si una parte tiene como antecesor q
 - ▶ Y otra no
- Se para cuando ya no se puede partir más
 - ▶ Las clases que contienen un sólo estado no se pueden partir
 - ▶ Si hemos mirado una clase y no se parte, ya no se parte más (se quita de la lista de trabajo)
 - ▶ Sólo los nuevos o los que no hemos mirado hay que re-partirlos
- Complejidad asintótica $kn \log(n)$ con k la longitud del alfabeto y n el número de estados

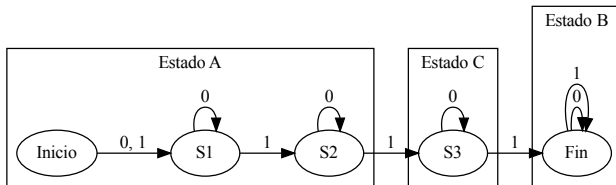
Minimización DFA: Hopcroft

- Empiezo por el estado A (clase de equivalencia inicial)
- Con símbolo 1, divido el estado A en predecesores de A y no predecesores de A
- Con símbolo 0 van todos al mismo



Minimización DFA: Hopcroft

- Empiezo por el estado A
- Con símbolo 1, parto A en $\{S3\}$, $\{Inicio, S1, S2\}$
- Con símbolo 0 van todos al mismo
- Y así sigo... (el ejemplo está sin acabar)



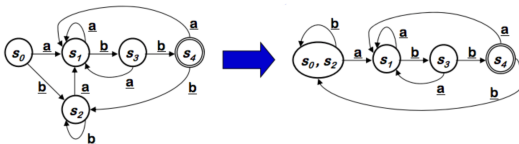
Minimización DFA: Hopcroft

- Último paso, cada clase de equivalencia es un sólo estado
- Todas las transiciones internas de la clase de equivalencia son de vuelta al mismo estado

Minimización DFA: Hopcroft

- Ejemplo completo

	Particion actual	lista trabajo	s	partir en a	partir en b
P_0	$\{s_4\} \{s_0, s_1, s_2, s_3\}$	$\{s_4\}$ $\{s_0, s_1, s_2, s_3\}$	$\{s_4\}$	no	$\{s_0, s_1, s_2\}$ $\{s_3\}$
P_1	$\{s_4\} \{s_3\} \{s_0, s_1, s_2\}$	$\{s_0, s_1, s_2\}$ $\{s_3\}$	$\{s_3\}$	no	$\{s_0, s_2\} \{s_1\}$
P_2	$\{s_1\} \{s_0, s_2\} \{s_4\} \{s_3\}$	$\{s_0, s_2\} \{s_1\}$	$\{s_1\}$	no	no



Libre de contexto

- Tipo-2
- $A \rightarrow \gamma$
- Autómata con pila (Push Down Automaton, PDA)
- A no terminal, γ terminal o no terminal

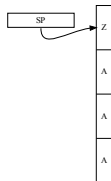
- Un PDA es un autómatas aumentado con una pila
- En la función de transición del estado $S1$ al $S2$ tiene $a, b \rightarrow z$
- Sigue la transición si la entrada es a , el símbolo en la cima de la pila es b
- En ese caso, haz pop de b y push de z

Definición PDA

- Autómata como el de antes NDFA, por ejemplo
- Conjunto de estados Q
- Función de transición $\delta : Q \times R \rightarrow Q$
- Lenguaje de entrada, terminales $\Sigma = \{a, b, c\}$
- Lenguaje de la pila, (no tienen por qué ser los terminales)
 $\Gamma = \{A, Z, R\}$
- Estado inicial de la pila $\Phi = (X, Z)$
- Estado inicial *inicio* $\in Q$
- Estados aceptadores $F \subseteq Q$

PDA

- Aparte de la definición del autómata tengo un language de pila, por ejemplo $\Gamma = \{A, Z \dots\}$: no tiene por qué ser el de entrada, Σ .
- Aparte de la definición del autómata tengo un estado inicial de la pila, que es una tupla de símbolos (en nuestra notación, la cima a la derecha) por ejemplo $\Phi = (A, A, A, Z)$ con $\Phi_i \in \Gamma$
- Sería equivalente a una pila vacía a la que se le ha hecho $push(A)$, $push(A)$, $push(A)$, $push(Z)$ en ese orden.



Ejemplo PDA

- El lenguaje es $\{\epsilon, 01, 0011, 000111 \dots\}$
- El mismo número de 0s que de 1s
- No es regular (aplicación de el Lema de bombeo, *pumping lemma*)

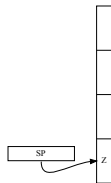
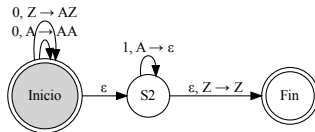
https://en.wikipedia.org/wiki/Pumping_lemma_for_regular_languages

Rabin, Michael; Scott, Dana (Apr 1959). "Finite Automata and Their Decision Problems". IBM Journal of Research and Development. 3 (2): 114–125. doi:10.1147/rd.32.0114. Lemma 8, p.119

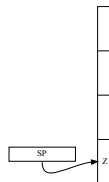
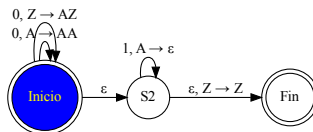
Ejercicio 3.2 del libro del dragón

Ejemplo PDA

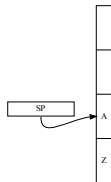
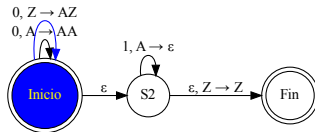
- Lenguaje de entrada $\Sigma = \{0, 1\}$
- Lenguaje de la pila $\Gamma = \{A, Z\}$
- Estado inicial de la pila $\Phi = (Z)$
- Estado de inicio *Inicio*
- Estado de aceptación *Fin*



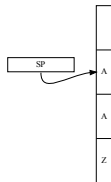
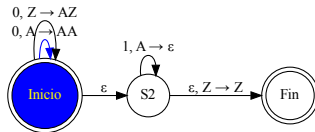
$0^n 1^n, n \geq 0$



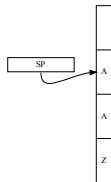
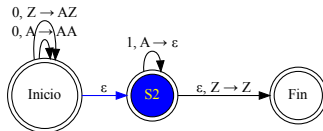
Entrada 0011



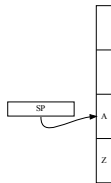
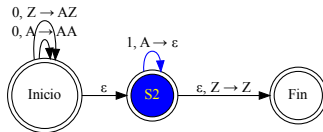
Entrada **0**₀₁₁



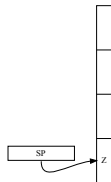
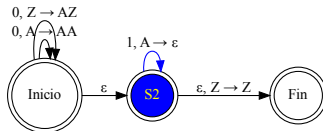
Entrada 0011



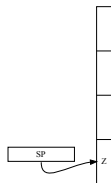
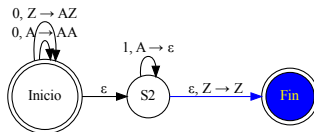
Entrada 00**1**1



Entrada 00**1**1



Entrada 001**1**



Entrada 001**1**✓

Lenguajes dependientes de contexto

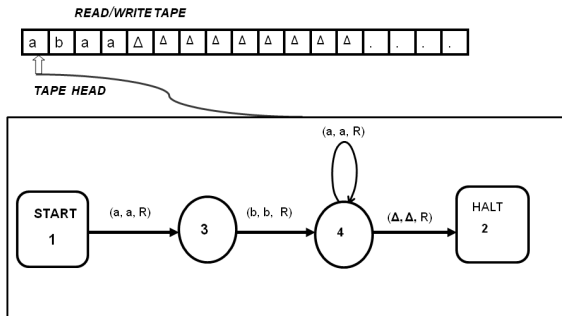
- El lenguaje de arriba es libre de contexto, pero
- $0^n 1^n 2^n$ o $0^n 1^m 2^n 3^m$, similar a declaraciones es dependiente de contexto (no vale con un autómata con pila)
- Estrategia similar a la anterior pero con varias pilas (más poderoso de lo necesario)

Máquinas de Turing

- Abstracción de computación (se dice que un lenguaje de programación es Turing-completo. . .)
- Máquina de estados con una cinta, con un puntero actual
- Las operaciones son Derecha (R), Izquierda (L), No mover (S)
- Es similar al autómatas con pila, me encuentro algo, escribo algo, muevo el puntero (pero hay más libertad para moverlo)
- El símbolo vacío se suele escribir como Δ
- Las transiciones, igual que en el autómatas, dependen del puntero de la cinta
- La entrada está en la cinta (hay estado de aceptación, pero la entrada, el programa y los datos son la cinta)
- Universal: la función de transición de estado, tabla en la cinta, *programa = datos*

Máquinas de Turing

Ejemplo. Ojo, los datos de entrada están en el estado inicial de la cinta.



A Turing Machine for aba^*

Dependiente de contexto

- Tipo-1
- $\alpha A \beta \rightarrow \alpha \gamma \beta$
- Lo reconoce un autómata lineal acotado (Linear Bounded Automaton, LBA), máquina de Turing acotada (Cinta finita)
- $\alpha \beta$ terminal o no terminal o ϵ
- A no terminal, γ terminal o no terminal

Enumerable recursivo

- Tipo-0, sin restricciones
- Cualquier regla de gramática $\alpha \rightarrow \beta$
- $\alpha\beta$ terminal, no terminal o ϵ
- Lo reconoce una Máquina de Turing (universal) o un autómata con 2 pilas (equivalente)

Expresiones regulares

- Subconjunto muy útil dentro de los lenguajes regulares/autómatas finitos
- Rápido y polinómico (expresiones regulares extendidas de Unix)
- De la expresión regular se construye un NFA y se simula navegando en paralelo

Russ Cox (2007) "[Regular Expression Matching Can Be Simple And Fast](#)"

Ken Thompson, "[Regular expression search algorithm](#)," Communications of the ACM 11(6) (June 1968), pp. 419–422.

Expresiones regulares/autómatas

- A partir de ejemplos se pueden aprender (algoritmo L^*)
- Muy útil, ejemplo de uso, reversing cachés de intel

Dana Angluin, (1987) "[Regular expression search algorithm](#)," Information and Computation 75, 87-106 Pepe Vila, Pierre Ganty, Marco Guarnieri, Boris Köpf, (2019) "[CacheQuery: Learning Replacement Policies from Hardware Caches](#)," arXiv Bollig B, Habermehl P, Kern C, Leucker M, (2009) "[Angluin-Style Learning of NFA](#)," InTwenty-First International Joint Conference on Artificial Intelligence Jun 26

Tema 1: Scanner/Lexer, tokens

Autómatas y desarrollo avanzado de software

Gorka Guardiola Múzquiz

GSYC

23 de noviembre de 2022



Lenguaje formal: terminales

- Usar como símbolos terminales caracteres, posible, pero poco legible
- Se usan “palabras” del lenguaje
- Tokens

Lexing/Scanning (tokenization)

- El primer paso de reconocer un lenguaje (compilador)
- Reconocer tokens, palabras (análisis léxico)
- Se suele describir con un lenguaje regular (tipo 3)
- Sencillo, se puede hacer con expresiones regulares (lex)
- Al final es una máquina de estados, fácil de programar a mano, mucho menos pesado

Tokens

// You can edit this code!
// Click here and start typing.

package main

import "fmt"

func main

fmt.Println("Hello, 世界")

Tokens

- Ignora comentarios, espacios. . .
- Convierte una lista de caracteres en tokens
- Anota los tokens con fichero y número de línea

Tokens

- Un token son 3 cosas:
 - ▶ Lexema, para depurar, trozo de texto aceptado
 - ▶ Tipo (el lenguaje formal se define en base a esto)
 - ▶ Valor

- Ejemplo 0x2, 2

- ▶ Lexema: “0x2”, “2”
- ▶ Tipo: ValInt (en ambos casos)
- ▶ Valor: 2 (en ambos casos)
- ▶ Posición (para errores, fichero, número de línea...)

Tokens

- Ejemplo "Hello world\x21"
 - ▶ Lexema: ""Hello world\x21""
 - ▶ Tipo: ValString
 - ▶ Valor: Hello world!
 - ▶ Posición x.go:28

Tokens

- Este programa se convierte en:

```
1 // Comentario
2 // Mas comentario
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello")
7 }
```

Tokens

- (“import”, TImport, -), (“fmt”, TStringLit, “fmt”), (“func”, TFunc, -), (“main”, TId, main), ((, TLPare, -)

```
1 // Comentario
2 // Mas comentario
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello")
7 }
```

Tokens

- Valores de tipos (int, char...)
- Palabras reservadas (int, char, func, import...)
- Identificadores (nombres definidos por el usuario)
- Puntuación (Paréntesis, llaves, final de línea)
- Literales ("bla bla", 32, 'c', ...)
- Operadores (Asignación, suma...)
- Depende mucho del lenguaje, por ejemplo en Python la tabulación cuenta y son tokens

Tokens

- Se tiene que poder reconocer el tipo mediante una expresión regular o en una tabla
- El orden de búsqueda en la tabla es importante (miro si es una palabra reservada y si no es un identificador)
- Típicamente, será una máquina de estados para reconocer pocas categorías (id, literales varios, puntuación) y luego completar (reservadas)

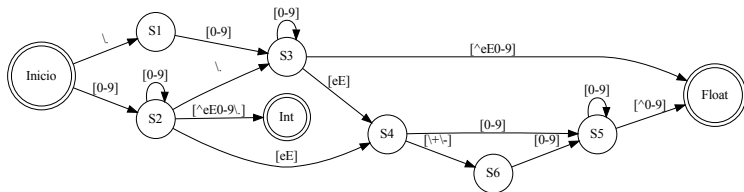
Tokens

- Ojo al definir el lenguaje, no hay ámbitos/contextos, o hago todo más complicado
- Si `>>` significa varias cosas diferentes (`C++`), tengo un problema.
- `>>` en `C++` es shift, es escribir en un stream, puntuación de una Template *Vector* `< Vector < Number >>`
- Definir significados únicos si es posible

Tokens: ejemplo, número en coma flotante

- .3 4.7 4.5e - 16
- +3.5 son dos tokens (operador signo y valor float)
- . - 9 son tres tokens (DOT MINUS INTVAL:9) y puede o no ser un error, pero no lo pilla el lexer
- La expresión regular sería:
$$([0-9]^*\backslash.[0-9] + ([eE][+\-]?[0-9]^+)?[^\-9])|$$
$$([0-9] + (\backslash.[0-9]^+)?[eE][+\-]?[0-9] + [^\-9])$$

Tokens: ejemplo, número en coma flotante (y de paso enteros)



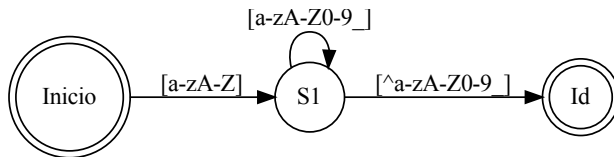
Tokens: ejemplo

- Si hay cualquier entrada que no es la que se espera, error (en el lexer)
- Si es una entrada que no pertenece pero está en la máquina de estados, acaba el token

Tokens: ejemplo, identificador

- Una letra seguida de varios caracteres que pueden ser una letra, un número o `_` (subrayado)
- La expresión regular sería: $[a-zA-Z][a-zA-Z0-9_]*$

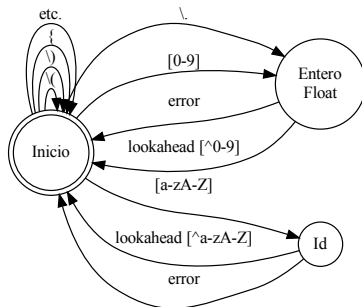
Tokens: ejemplo, identificadores



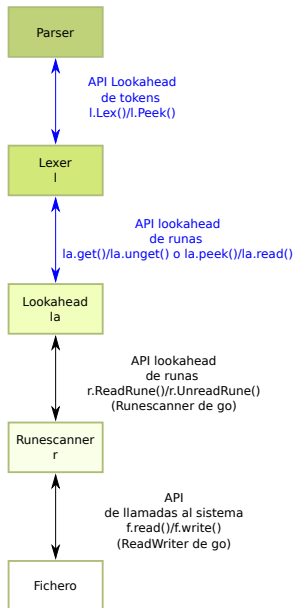
Tokens: ejemplo, tokenizador de un lenguaje

- Jerarquía de máquinas de estados
- Las submáquinas de estados salen cuando encuentran algo que no reconocen
- Miran en siguiente (lookahead), si no es para ellas, vuelven
- En general cuando hay que desambiguar se mira el siguiente (lookahead), y se elige un camino
- No se consume un token hasta que se sabe con certeza
- Tiene salida (token aceptado) en las transiciones de entrada a Inicio

Tokens: jerarquía



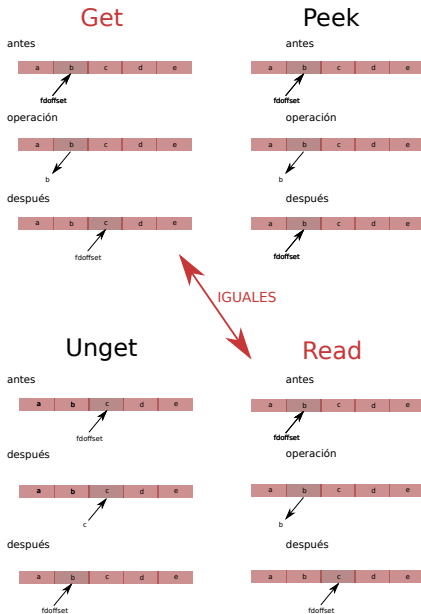
Lexer: APIs



Tokens

- Vamos a ir carácter a carácter, el buffering es imprescindible
- Queremos poder ver el siguiente carácter (Lookahead)
- Dos APIs posibles, leer y devolver (get/unget) o vistazo y leer (peek/read)

Tokens



Tokens

- Usando bufio (API tipo Get, UnGet)
- Define el interfaz RuneScanner

```
1 type RuneScanner interface {  
2     ReadRune() (r rune, size int, err error)  
3     UnreadRune() error  
4 }  
5  
6 type Lexer struct {  
7     file string  
8     line int  
9     r RuneScanner  
10    lastrune rune  
11 }
```

Lexer

- Usando bufio
- Se le puede pasar ya un RuneScanner (siguiente)

```
1 func NewLexer(file string) (l *Lexer, err error) {  
2     l = &Lexer{line: 1}  
3     f, err := os.Open(file)  
4     if err != nil {  
5         return nil, err  
6     }  
7     l.file = file  
8     l.r = bufio.NewReader(f)  
9     return l, nil  
10 }
```

Lexer

- Se le puede pasar ya un RuneScanner

```
1 func NewLexer(r RuneScanner, fname string) (l *Lexer, err error) {  
2     l = &Lexer{line: 1}  
3     l.file = fname  
4     l.r = r  
5     return l, nil  
6 }
```

Lexer: un API de lookahead

- get y unget
- Llamar a unget (igual que UnreadRune) es un error

```
1 func (l *Lexer) get() (r rune, err error) {
2     r, _, err = l.r.ReadRune()
3     if err == nil {
4         l.lastrune = r
5         if r == '\n' {
6             l.line++
7         }
8     }
9     return r, err
10 }
11
12 func (l *Lexer) unget() (err error) {
13     err = l.r.UnreadRune()
14     if err == nil && l.lastrune == '\n' {
15         l.line--
16     }
17     return err
18 }
```

Lexer: otro API, peek y get (no vamos a usar)

```
1 func (l *Lexer) peek() (r rune, err error) {
2     lastr := l.lastrune
3     r, _, err = l.r.ReadRune()
4     if err != nil {
5         return r, err
6     }
7     l.r.UnreadRune()
8     l.lastrune = lastr
9     return r, err
10 }
11 func (l *Lexer) get() (r rune, err error) {
12     r, _, err = l.r.ReadRune()
13     if err == nil {
14         l.lastrune = r
15     }
16     if r == '\n' {
17         l.line++
18     }
19     return r, err
20 }
```


Lexer: errores fatales

- Si hay errores, es grave, tengo que acabar
- No son errores de tokenización, sino de codificación/problemas con el fichero...

```
1 func (l *Lexer) get() (r rune) {
2     var err error
3     r, _, err = l.r.ReadRune()
4     if err == nil {
5         l.lastrune = r
6         if r == '\n' {
7             l.line++
8         }
9     }
10    if err == io.EOF {
11        l.lastrune = RuneEof
12        return RuneEof
13    }
14    if err != nil {
15        panic(err)
16    }
17    return r
18 }
```

Lexer: errores fatales

```
19 func (l *Lexer) unget() {  
20     var err error  
21     if l.lastrune == RuneEof {  
22         return  
23     }  
24     err = l.r.UnreadRune()  
25     if err == nil && l.lastrune == '\n' {  
26         l.line—  
27     }  
28     if err != nil {  
29         panic(err)  
30     }  
31 }
```

Lexer: errores fatales

- En main recojo el panic, vuelco la pila solo en depuración
- Aprovecho para reescribir los errores de acceso de memoria y demás

```
1  const (  
2      BugMsg = "compiler error:"  
3      RunMsg = "runtime error:"  
4  )  
5  defer func() {  
6      if r := recover(); r != nil {  
7          errs := fmt.Sprintf(r)  
8          if strings.HasPrefix(errs, "runtime error:") {  
9              errs = strings.Replace(errs, RunMsg, BugMsg, 1)  
10         }  
11         err = errors.New(errs)  
12         if Dflag {  
13             fmt.Fprintf(os.Stderr, "%s\n%s", err, debug.Stack())  
14         }  
15     }  
16 }()
```

Lexer: acumulación de lexema

- Voy a ir leyendo y guardándolo y eso va a ser el lexema
- Lo guardo implícitamente cuando llamo a get (y devuelvo en unget si hay)

Lexer: acumulación de lexema

- El lexema que llevo aceptado va en `accepted`

```
1 type Lexer struct {  
2     file string  
3     line int  
4     r RuneScanner  
5     lastrune rune  
6  
7     accepted [] rune  
8 }
```

Lexer: acumulación de lexema

- Lo guardo en get

```
1 func (l *Lexer) get() (r rune) {
2     var err error
3     r, _, err = l.r.ReadRune()
4     if err == nil {
5         l.lastrune = r
6     }
7     if r == '\n' {
8         l.line++
9     }
10    if err == io.EOF {
11        l.lastrune = RuneEof
12        return RuneEof
13    }
14    if err != nil {
15        panic(err)
16    }
17    l.accepted = append(l.accepted, r)
18    return r
19 }
```

Lexer: acumulación de lexema

- Lo quito en unget

```
20 func (l *Lexer) unget() {
21     var err error
22     if l.lastrune == RuneEof {
23         return
24     }
25     err = l.r.UnreadRune()
26     if err == nil && l.lastrune == '\n' {
27         l.line —
28     }
29     l.lastrune = unicode.ReplacementChar
30     if len(l.accepted) != 0 {
31         l.accepted = l.accepted[0: len(l.accepted) - 1]
32     }
33     if err != nil {
34         panic(err)
35     }
36 }
```

Lexer: aceptar

- El lexema que llevo aceptado va en `accepted`
- Cuando llega el estado adecuado, lo acepto

```
1 func (l *Lexer) accept() (tok string){
2     tok = string(l.accepted)
3     if tok == "" && l.lastrune != RuneEof {
4         panic(errors.New("empty token"))
5     }
6     l.accepted = nil
7     return tok
8 }
```


Lexer: aceptar

- Ahora ya estamos listos para el interfaz que tendrá el Lexer desde fuera
- Queremos poder pedir el siguiente token
- Ya estamos preparados para construir la jerarquía de máquinas de estados
- Cada submáquina es una función independiente

Lexer: jerarquía (una calculadora)

- Ojo, truco interesante, las runas son enteros
- Para los paréntesis, etc. (tokens de una sólo runa) uso de tipo el valor de la runa
- Uso el rango de runas inválidas para el resto
- Hack para hacerlo mas cómodo

Lexer: jerarquía (una calculadora)

```
1 type TokenType rune
2
3 const (
4     RuneEof = unicode.MaxRune + 1 + iota
5     TokFloatVal          //careful, if there are too many, there has to be space
6     TokId
7     TokEol = TokenType('\n')
8     TokLPar = TokenType('(')
9     TokRPar = TokenType(')')
10    TokAdd  = TokenType('+')
11    TokMul  = TokenType('*')
12    TokMin  = TokenType('-')
13    TokDiv  = TokenType('/')
14    TokEq   = TokenType('=')
15    TokEof  = TokenType(RuneEof)
16    TokBad   = TokenType(0) //this zero so uninitialized is bad
17 )
18
19 type Token struct {
20     Lexema    string
21     Type      TokenType
22     FloatVal  float64
23 }
```

Lexer: jerarquía (una calculadora)

```
1 func (l *Lexer) Lex() (t Token, err error) {
2
3     for r := l.get(); ; r = l.get() {
4         if unicode.IsSpace(r) && r != '\n' {
5             l.accept()
6             continue
7         }
8         switch r {
9             case '(', ')', '*', '/', '+', '-':
10             t tokType = TokType(r)
11             t.lexema = l.accept()
12             return t, nil
13             case RuneEof:
14             t tokType = TokEof
15             l.accept()
16             return t, nil
17             case '\n':
18             t tokType = TokEol
19             t.lexema = l.accept()
20             return t, nil
21         }
```

Lexer: jerarquía

```
22  switch {
23  case r == '.', unicode.IsDigit(r): //Number
24      l.unget()
25      t, err = l.lexNum()
26      return t, err
27  case unicode.IsLetter(r):
28      l.unget()
29      t, err = l.lexId()
30      return t, err
31  default:
32      errs := fmt.Sprintf("bad rune%c: %x", r, r)
33      return t, errors.New(errs)
34  }
35 }
36 return t, err
37 }
```

Lexer: particular del lenguaje

- ¿Es el un token el espacio?
- ¿Es un token el final de fichero?
- Delimita final de línea o punto y coma o...

Lexer: identificador (ojo con unicode)

```
1 func (l *Lexer) lexId() (t Token, err error) {
2     r := l.get()
3     if !unicode.IsLetter(r) {
4         return t, errors.New("bad Id, should not happen")
5     }
6     isAlpha := func(ar rune) bool {
7         return unicode.IsDigit(ar) || unicode.IsLetter(ar) || r == '_'
8     }
9     for r = l.get(); isAlpha(r); r = l.get() {
10    }
11    l.unget()
12    t.tokType = TokId
13    t.lexema = l.accept()
14    return t, nil
15 }
```

Lexer: un número

```
1 func (l *Lexer) lexNum() (t Token, err error) {
2     const (
3         Es      = "Ee"
4         Signs = "+-~"
5     )
6     hasDot := false
7     r := l.get()
8     if r == '.' {
9         hasDot = true
10        r = l.get()
11    }
12    for ; unicode.IsDigit(r); r = l.get() {
13    }
14    if r == '.' {
15        if hasDot {
16            return t, errors.New("bad float [" + l.accept() + "]")
17        }
18        hasDot = true
19        for r = l.get(); unicode.IsDigit(r); r = l.get() {
20        }
21    }
```


Lexer: un número

```
22  switch {
23  case strings.ContainsRune(Es, r):
24      r = l.get()
25      if strings.ContainsRune(Signs, r) {
26          r = l.get()
27      }
28  case hasDot:
29      l.unget()
30      break
31  case !hasDot: //may be an int
32      l.unget()
33      t.lexema = l.accept()
34      t.tokIntVal, err = strconv.ParseInt(t.lexema, 10, 64)
35      if err != nil {
36          return t, errors.New("bad int [" + t.lexema + "]")
37      }
38      t.tokType = TokIntVal
39      return t, nil
40  default:
41      return t, errors.New("bad float [" + l.accept() + "]")
42  }
```

Lexer: un número

```
43  for r = l.get(); unicode.IsDigit(r); r = l.get() {  
44  }  
45  l.unget()  
46  t.lexema = l.accept()  
47  //estoy usando ParseFloat, me vale el de go?  
48  t.tokFloatVal, err = strconv.ParseFloat(t.lexema, 64)  
49  if err != nil {  
50      return t, errors.New("bad float [" + t.lexema + "]")  
51  }  
52  t.tokType = TokFloatVal  
53  return t, nil  
54 }
```

Lexer: lookahead

- Necesito también poder hacer lookahead de tokens
- De nuevo, como get/peek get/unget
- Guardo en el lexer el siguiente token
- Peek llama a Lex si no hay salvado, salva el token y devuelve el salvado pero no lo borra
- Lex devuelve el salvado y lo borra si lo hay o busca uno nuevo (lo que hacía antes Lex)

Lexer: lookahead

```
1 type Lexer struct {  
2     file      string  
3     line      int  
4     r         RuneScanner  
5     lastrune  rune  
6  
7     accepted  [] rune  
8     tokSaved  *Token  
9 }
```

Lexer: lookahead

```
10 func (l *Lexer) Peek() (t Token, err error) {
11     t, err = l.Lex()
12     if err == nil {
13         l.tokSaved = &t
14     }
15     return t, nil
16 }
17
18 func (l *Lexer) Lex() (t Token, err error) {
19     if l.tokSaved != nil {
20         t = *l.tokSaved
21         l.tokSaved = nil
22         return t, nil
23     }
24     for r := l.get(); ; r = l.get() {
25         if unicode.IsSpace(r) && r != '\n' {
26             l.accept()
27             continue
28         }
29         ...
```

Lexer: tests

- Se puede probar `lexNum()`, `lexId()` por separado (caja blanca)
- Todo el lexer con un reader (otro `NewLexer(r RuneScanner)`)
- Se recubre una string que representa el contenido del fichero (`strings.NewReader`)
- Con un `RuneScanner` (`get/unget`) que pierde runas, modifica, etc. (fuzzing para el lexer)
- Para luego: envolver el lexer con otro lexer que pierde tokens o los modifica al azar (fuzzing para el parser)

Lexer: palabras reservadas

- Después de `lexId()` se mira en una tabla hash (diccionario)
- Se devuelve el token adecuado
- Tipo de token: COS, IF, ELSE, FOR... (sin valor, Nada)

Lexer: tipos de datos

- Después de `lexId()` se mira en una tabla hash (diccionario)
- Tipo de token: `TokType`, valor: `TypeInt`, `TypeBool`...
- Que sigan los mismos mecanismos (misma tabla) que los definidos por usuario (si hay una tabla de tipos, la misma para los dos)
- Para hacerlo igual que los definidos por el usuario, el lexer tiene que poder mirar en la tabla de símbolos (más adelante) y devolver símbolos en lugar de tokens, esto es útil en general para ayudar con la gramática (`VarName` vs `Id`)
- The lexer hack
https://en.wikipedia.org/wiki/The_lexer_hack

Lexer: tipo de token vs. valor

- Siempre existe la posibilidad de poner codificar de una manera u otra, pero una mejor (viendo el parser)
- Ejemplo el lexema `int` podría corresponderse con un token de tipo `TokInt` y valor: Nada o de tipo `TokType` y de valor `TokInt`
- El tipo de token (o de símbolo más adelante) tiene que ver con lo que aparece en la gramática
- Hay que ser cuidadoso con esto o me aparecerán muchas más reglas de la gramática (o no podré codificar precedencia...)
 - ▶ *Decl ::= Type Var ';' ;*
 - ▶ Mejor que
 - ★ *Decl ::= IntType Var ';' ;*
 - ★ *Decl ::= BoolType Var ';' ;*
 - ★ *Decl ::= CharType Var ';' ;*
 - ★ ...

Lexer: tipos de datos, builtins...

- Se inicializa la tabla de símbolos al arrancar como si ya hubiese cosas definidas
- Lo veremos cuando veamos contextos (más adelante)

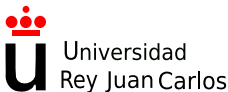
Tema 2: Análisis sintáctico

Autómatas y desarrollo avanzado de software

Gorka Guardiola Múzquiz

GSYC

23 de noviembre de 2022



Análisis sintáctico

- El analizador sintáctico también se llama *parser* (analizar: parsing)
- Construye árbol sintáctico, AST (Abstract Syntax Tree)
- Estructura de datos en árbol
- Siguiendo la gramática
- Los terminales de la gramática son los tokens resultado del análisis léxico

Análisis sintáctico

- Primero definimos la gramática (libre de contexto) para el lenguaje
- Hay restricciones que dependen de la implementación (LR. . . , veremos más adelante)

Gramáticas libre de contexto

- Tipo-2
- $A \rightarrow \gamma$
- A no terminal, γ cadena de terminal o no terminal
- **No puede haber nada extra con A en el lado izquierdo,**
prohibidas producciones del tipo $bAc \rightarrow \gamma$

Gramática para una calculadora

$$\text{Expr} \rightarrow \text{Expr} \text{'*'} \text{Expr}$$
$$\text{Expr} \rightarrow \text{Expr} \text{'+'} \text{Expr}$$
$$\text{Expr} \rightarrow \text{Expr} \text{'-'} \text{Expr}$$
$$\text{Expr} \rightarrow \text{Expr} \text{'/'} \text{Expr}$$
$$\text{Expr} \rightarrow \text{'(' Expr ')')}$$
$$\text{Expr} \rightarrow \text{num}$$

- He puesto los terminales que son un char entre comillas

Notación BNF

- Backus-Naur Form, Backus Normal Form
- Notación estándar (no terminales entre menor y mayor. . .)

```
<EXPR> ::= <EXPR> '*' <EXPR> |  
          <EXPR> '+' <EXPR> |  
          <EXPR> '-' <EXPR> |  
          <EXPR> '/' <EXPR> |  
          '(' <EXPR> ')' |  
          num
```


Derivación

- Parto de símbolo inicial `EXPR`
- Reemplazamos el primer no terminal a la izquierda
- Aplicando una regla de producción tras otra
- Hasta que sólo haya terminales
- Esto es una derivación

Ejemplo derivación

- $(3 + 5) * 2$

-> `EXPR`

-> `EXPR * EXPR`

-> `( EXPR ) * EXPR`

-> `(EXPR + EXPR) * EXPR`

-> `(3 + EXPR) * EXPR`

-> `(3 + 5) * EXPR`

-> `(3 + 5) * 2`

Ejemplo derivación

- Se dice que $EXPR \xrightarrow{*} (3 + 5) * 2$
- La derecha deriva de la izquierda en uno o más pasos
- Si hay una derivación, la cadena pertenece al lenguaje

Ejemplo derivación

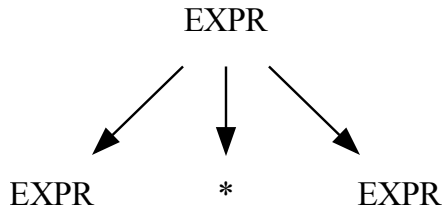
- Lo que hacemos es construir un árbol sintáctico abstracto (Abstract Syntax Tree: AST)
- En cada derivación va creciendo hasta que todas las hojas son no terminales

-> `EXPR`

`EXPR`

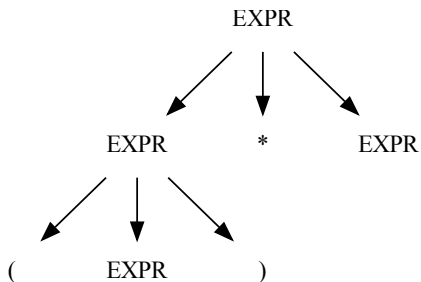
AST

-> `EXPR * EXPR`



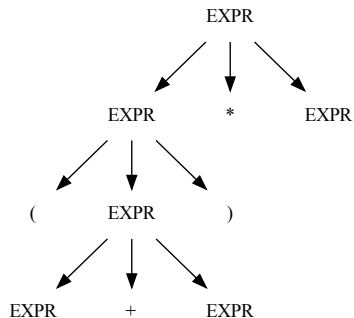
AST

-> (EXPR) * EXPR



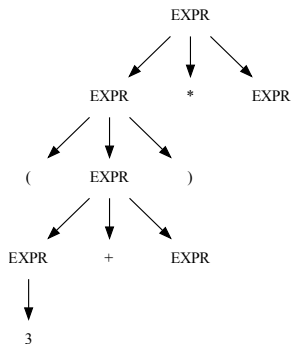
AST

-> (EXPR + EXPR) * EXPR



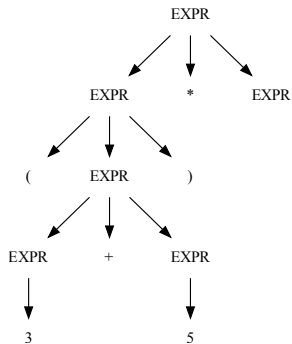
AST

-> (3 + EXPR) * EXPR



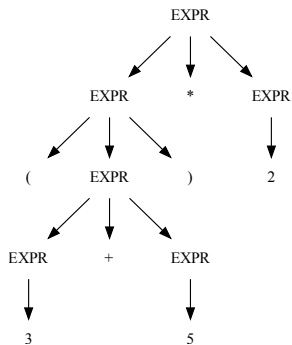
AST

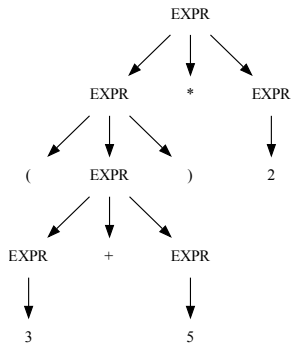
-> (3 + 4) * EXPR



AST

-> (3 + 4) * 2





AST

- La raíz es el inicial
- Terminales son las hojas
- No terminales son los nodos intermedios

Derivación

- Hemos ido sustituyendo no terminales por la izquierda (Left-most)
- Podríamos hacerlo al revés (Right-most)
- Importante más adelante para la implementación
- **El árbol es el mismo**, pero las reglas de producción de la gramática se aplican en otro orden

Ambigüedad

- Tenemos varias reglas de producción que se pueden aplicar
- El resultado son árboles distintos
- Pueden o no ser semánticamente equivalentes

Ejemplo derivación

- La gramática anterior es ambigua, dos derivaciones

- $3 + 5 * 2$

-> `EXPR`

-> `EXPR * EXPR`

-> `EXPR + EXPR * EXPR`

-> `3 + EXPR * EXPR`

-> `3 + 5 * EXPR`

-> `3 + 5 * 2`

Ejemplo derivación

- Otra derivación (parece igual pero el orden es diferente)
- $3 + 5 * 2$

-> `EXPR`

-> `EXPR + EXPR`

-> `EXPR + EXPR`

-> `EXPR + EXPR * EXPR`

-> `EXPR + EXPR * 2`

-> `EXPR + 5 * 2`

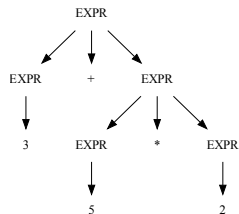
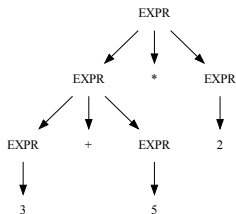
-> `3 + 5 * 2`

Ambigüedad

- ¿Qué aplico primero?
- La regla $\text{EXPR} + \text{EXPR}$ ó $\text{EXPR} * \text{EXPR}$
- Uno: $(3 + 5) * 2 = 16$
- El otro: $3 + (5 * 2) = 13$

Ambigüedad

- $(3 + 5) * 2$
- $3 + (5 * 2)$



Ambigüedad

- La gramática y por tanto el lenguaje es ambiguo
- Existe más de una derivación left-most (o right-most)
- La cadena de entrada no tiene sentido
- Se puede arreglar la gramática
- Una solución es cambiar las reglas de derivación

Ambigüedad

$\langle \text{EXPR} \rangle ::= \langle \text{FACT} \rangle '+' \langle \text{EXPR} \rangle \mid$
 $\langle \text{FACT} \rangle '-' \langle \text{EXPR} \rangle \mid$
 $\langle \text{FACT} \rangle$

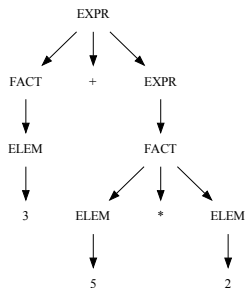
$\langle \text{FACT} \rangle ::= \langle \text{ELEM} \rangle '*' \langle \text{FACT} \rangle \mid$
 $\langle \text{ELEM} \rangle '/' \langle \text{FACT} \rangle \mid$
 ELEM

$\langle \text{ELEM} \rangle ::= \text{num} \mid$
 $'(' \langle \text{EXPR} \rangle ')'$

- Dejamos para el final el operador que más precedencia tiene (los productos)
- Hacemos primero la operación asociada al operador que menos precedencia tiene (la sumas)
- Sigue siendo ambigua $+$ vs $-$ pero no importa
- Sigue siendo ambigua $a/b * c$ y esto sí importa, podemos seguir

Ambigüedad

- Ahora $3+5*2$



Gramática no ambigua

$\langle \text{EXPR} \rangle ::= \langle \text{DIVI} \rangle \text{'+' } \langle \text{EXPR} \rangle \mid$
 $\langle \text{DIVI} \rangle \text{'-' } \langle \text{EXPR} \rangle \mid$
 $\langle \text{DIVI} \rangle$

$\langle \text{DIVI} \rangle ::= \langle \text{FACT} \rangle \text{'/' } \langle \text{DIVI} \rangle \mid$
 FACT

$\langle \text{FACT} \rangle ::= \langle \text{DIVI} \rangle \text{'*'} \langle \text{FACT} \rangle \mid$
 ELEM

$\langle \text{ELEM} \rangle ::= \text{num} \mid$
 $\text{'(' } \langle \text{EXPR} \rangle \text{' ')}$

- Ahora $a/b * c$ es $a/(b * c)$: quizás no hacía falta
- El caso que queda es ambiguo pero no importa (¿o sí?)
- Y si tengo $a * b * c$ ¿cual va primero?

Ambigüedad: dangling else

- Otro problema, similar al de operadores de antes
- Aparece en lenguajes con la construcción if/else

Dangling else

- Problema clásico (en los derivados de C, hay paréntesis y sentencias, si no hay llaves, sucede igual).

Parte de una gramática:

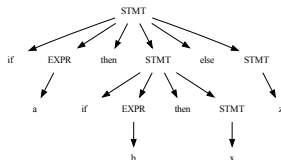
```
<STMT> ::= "if" <EXPR> "then" <STMT> |  
          "if" <EXPR> "then" <STMT> "else" <STMT> |  
          <OTHER>
```

```
1 if a then  
2     if b then  
3         x;  
4 else  
5     z;
```

```
1 if a then  
2     if b then  
3         x;  
4     else  
5         z;
```

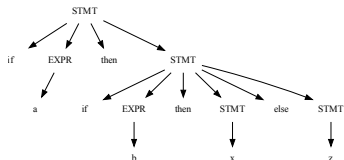
Ambigüedad

```
1 if a then
2     if b then
3         x;
4 else
5     z;
```



Ambigüedad

```
1 if a then
2   if b then
3     x;
4   else
5     z;
```



Dangling else

¿Qué debería pasar?

- El then más cercano sin cuerpo
 - ▶ Solución, reescribir la gramática para evitar ambigüedad
- Solución, cambiar el lenguaje, llaves o endif obligatorio (Go: llaves, o Pascal: **endif**)

Otra solución: ambigüedad

- Reescribir la gramática sin ambigüedad

```
<STMT> ::= <MATCHSTMT> |  
          <OPENSTMT>
```

```
<MATCHSTMT> ::= "if" <EXPR> "then" <MATCHSTMT> "else" <MATCHSTMT> |  
                <OTHER>
```

```
<OPENSTMT> ::= "if" <EXPR> "then" <STMT> |  
               "if" <EXPR> "then" <MATCHSTMT> "else" <OPENSTMT>
```

- Una sentencia entre then y else, si es un if, tiene que ser completa (matched)

Ambigüedad

- El caso de operadores, reescribir la gramática lo complica
- pasamos de una gramática sencilla a una bastante complicada

Ambigüedad

- De esto

```
<EXPR> ::= <EXPR> '*' <EXPR> |  
          <EXPR> '+' <EXPR> |  
          <EXPR> '-' <EXPR> |  
          <EXPR> '/' <EXPR> |  
          '(' <EXPR> ')' |  
          num
```

Ambigüedad

- A esto

$\langle \text{EXPR} \rangle ::= \langle \text{DIVI} \rangle \text{'+' } \langle \text{EXPR} \rangle \mid$
 $\langle \text{DIVI} \rangle \text{'-' } \langle \text{EXPR} \rangle \mid$
 $\langle \text{DIVI} \rangle$

$\langle \text{DIVI} \rangle ::= \langle \text{FACT} \rangle \text{'/' } \langle \text{DIVI} \rangle \mid$
 FACT

$\langle \text{FACT} \rangle ::= \langle \text{DIVI} \rangle \text{'*'} \langle \text{FACT} \rangle \mid$
 ELEM

$\langle \text{ELEM} \rangle ::= \text{num} \mid$
 $\text{'(' } \langle \text{EXPR} \rangle \text{' ')}$

Ambigüedad

- Sin embargo, con esto si pudiésemos elegir qué producción se aplica
- Cuando se pueden aplicar varias
- No habría ambigüedad

```
<EXPR> ::= <EXPR> '*' <EXPR> |  
          <EXPR> '+' <EXPR> |  
          <EXPR> '-' <EXPR> |  
          <EXPR> '/' <EXPR> |  
          '(' <EXPR> ')'  
          num
```

Ambigüedad

- Basta con asignar a cada operador un nivel de precedencia
- Y a los del mismo nivel decir si de izquierda a derecha ($a * b * c$) puede ser $(a * b) * c$ ó $a * (b * c)$
- Hemos cambiado la notación para la que usan los generadores de parsers (yacc)
- Cada línea un nivel de precedencia, left significa que asocian a izquierdas

```
%left '+' '-'
```

```
%left '*' '/'
```

```
EXPR ::= EXPR '*' EXPR |  
        EXPR '+' EXPR |  
        EXPR '-' EXPR |  
        EXPR '/' EXPR |  
        '(' EXPR ')' |  
        num
```

Ambigüedad

- $2 * 3 + 4 \rightarrow (2 * 3) + 4$
- $2 * 3/4/5 \rightarrow ((2 * 3)/4)/5$
- $2 + 3 - 4 + 5 \rightarrow (((2 + 3) - 4) + 5)$
- $2 + 3 * 4 + 5 \rightarrow ((2 + (3 * 4)) + 5)$

%left '+' '-'

%left '*' '/'

```
EXPR ::= EXPR '*' EXPR |  
        EXPR '+' EXPR |  
        EXPR '-' EXPR |  
        EXPR '/' EXPR |  
        '(' EXPR ')'  
num
```

Derivación

- Hay que tener una forma de elegir cual es la siguiente regla que se aplica (Left-most, Right-most)
- Con la gramática de antes:

```
<EXPR> ::= <EXPR> '*' <EXPR> |  
          <EXPR> '+' <EXPR> |  
          <EXPR> '-' <EXPR> |  
          <EXPR> '/' <EXPR> |  
          '(' <EXPR> ')' |  
          num
```

Derivación

- Left-most, como antes
- $(3 + 5) * 2$

-> `EXPR`

-> `EXPR * EXPR`

-> `( EXPR ) * EXPR`

-> `(EXPR + EXPR) * EXPR`

-> `(3 + EXPR) * EXPR`

-> `(3 + 5) * EXPR`

-> `(3 + 5) * 2`

Derivación

- Right-most
- $(3 + 5) * 2$

-> `EXPR`

-> `EXPR * EXPR`

-> `EXPR * 2`

-> `( EXPR ) * 2`

-> `(EXPR + EXPR) * 2`

-> `(EXPR + 5) * 2`

-> `(3 + 5) * 2`

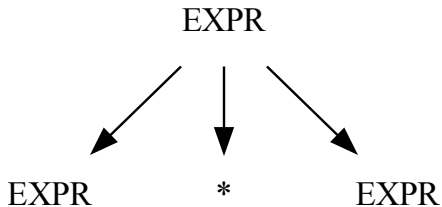
AST, right-most

-> `EXPR`

`EXPR`

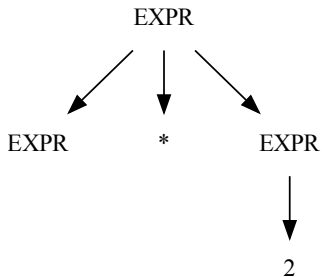
AST, right-most

-> $\text{EXPR} * \text{EXPR}$



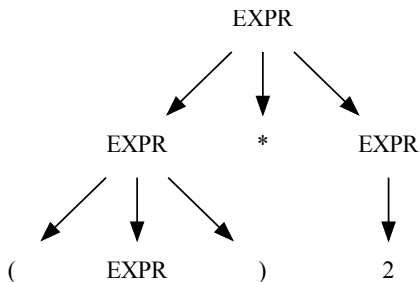
AST, right-most

-> `EXPR * 2`



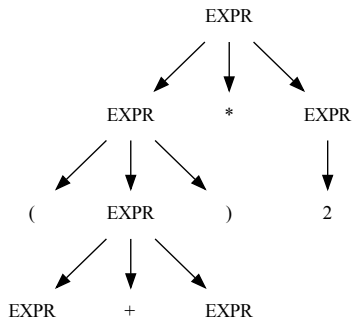
AST, right-most

-> (EXPR) * 2



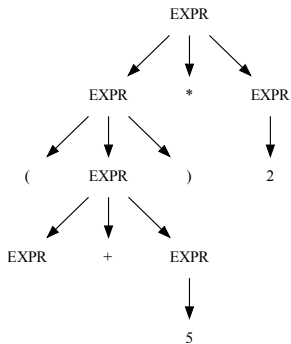
AST, right-most

-> (EXPR + EXPR) * 2



AST, right-most

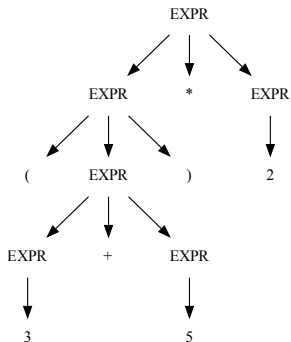
-> (EXPR + 5) * 2



AST, right-most

- El árbol final es el mismo

-> (3 + 5) * 2



Recursividad en gramáticas

- Forma de las reglas de producción (o se puede reescribir así)
- $A \rightarrow A XB$ recursiva por la izquierda
- $A \rightarrow B XA$ recursiva por la derecha
- Ejemplo, *EXPR* es recursivo por la derecha:

```
<EXPR> ::= <FACT> '+' <EXPR> |  
          <FACT> '-' <EXPR> |  
          <FACT>
```

```
<FACT> ::= <ELEM> '*' <FACT> |  
          <ELEM> '/' <FACT> |  
          ELEM
```

```
<ELEM> ::= num |  
          '(' <EXPR> ')'
```

Recursividad en gramáticas

- Una gramática puede ser recursiva en varios niveles/pasos (aplicando varias reglas)
- Ej. por la derecha sí $A \xrightarrow{*} X A$
- Ej. por la izquierda sí $A \xrightarrow{*} A X$
- Importante para la implementación: derivaciones left-most o right-most (extraer tokens vs bucle infinito)

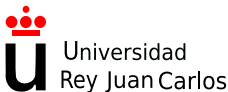
Tema 3: Parsing descendente recursivo

Autómatas y desarrollo avanzado de software

Gorka Guardiola Múzquiz

GSYC

23 de noviembre de 2022



Parsing descendente recursivo

- El objetivo es recorrer un AST (por ejemplo para construirlo)
- De la raíz (no terminales) a las hojas (terminales)
- Hay que ir eligiendo las reglas de producción según los tokens

Parsing descendente recursivo

- Comenzamos con una gramática sencilla
- Un programa es una lista de expresiones
- Se podrían usar otros delimitadores

$$\langle \text{PROG} \rangle ::= \langle \text{EXPR} \rangle \langle \text{PROG} \rangle \mid \\ \langle \text{EXPR} \rangle \langle \text{EOF} \rangle$$
$$\langle \text{EXPR} \rangle ::= \langle \text{EXPR} \rangle ' * ' \langle \text{EXPR} \rangle \mid \\ \langle \text{EXPR} \rangle ' + ' \langle \text{EXPR} \rangle \mid \\ ' (' \langle \text{EXPR} \rangle ') ' \mid \\ \text{num}$$

Parsing descendente recursivo

- Quitamos la ambigüedad
- Como vimos en el tema anterior

$$\begin{aligned} \langle \text{PROG} \rangle &::= \langle \text{EXPR} \rangle \langle \text{PROG} \rangle \mid \\ &\quad \langle \text{EXPR} \rangle \langle \text{EOF} \rangle \end{aligned}$$
$$\begin{aligned} \langle \text{EXPR} \rangle &::= \langle \text{FACT} \rangle \text{'+' } \langle \text{EXPR} \rangle \mid \\ &\quad \langle \text{FACT} \rangle \end{aligned}$$
$$\begin{aligned} \langle \text{FACT} \rangle &::= \langle \text{FACT} \rangle \text{'*'} \langle \text{ATOM} \rangle \mid \\ &\quad \langle \text{ATOM} \rangle \end{aligned}$$
$$\langle \text{ATOM} \rangle ::= \text{num} \mid \text{'(' } \langle \text{EXPR} \rangle \text{' '}$$

Backtracking

- Tiene un problema
- Imaginemos la expresión $3 * 5$
- Voy de izquierda a derecha (left-most)
- Leyendo tokens y aplicando reglas

$$\begin{aligned}\langle \text{PROG} \rangle &::= \langle \text{EXPR} \rangle \langle \text{PROG} \rangle \mid \\ &\quad \langle \text{EXPR} \rangle \langle \text{EOF} \rangle\end{aligned}$$
$$\begin{aligned}\langle \text{EXPR} \rangle &::= \langle \text{FACT} \rangle '+' \langle \text{EXPR} \rangle \mid \\ &\quad \langle \text{FACT} \rangle\end{aligned}$$
$$\begin{aligned}\langle \text{FACT} \rangle &::= \langle \text{FACT} \rangle '*' \langle \text{ATOM} \rangle \mid \\ &\quad \langle \text{ATOM} \rangle\end{aligned}$$
$$\langle \text{ATOM} \rangle ::= \text{num} \mid '(' \langle \text{EXPR} \rangle ')'$$

Backtracking

- El `.` indica lo que llevamos derivado
- `.3 * 5`

`<PROG> ::= <EXPR> <PROG> |`
`<EXPR> <EOF>`

`<EXPR> ::= <FACT> '+' <EXPR> |`
`<FACT>`

`<FACT> ::= <FACT> '*' <ATOM> |`
`<ATOM>`

`<ATOM> ::= num | '(' <EXPR> ')'`

Backtracking

- El . indica lo que llevamos derivado
- 3. * 5

->PROG

->EXPR

$\langle \text{PROG} \rangle ::= \langle \text{EXPR} \rangle \langle \text{PROG} \rangle \mid$
 $\langle \text{EXPR} \rangle \langle \text{EOF} \rangle$

$\langle \text{EXPR} \rangle ::= \langle \text{FACT} \rangle '+' \langle \text{EXPR} \rangle \mid$
 $\langle \text{FACT} \rangle$

$\langle \text{FACT} \rangle ::= \langle \text{FACT} \rangle '*' \langle \text{ATOM} \rangle \mid$
 $\langle \text{ATOM} \rangle$

$\langle \text{ATOM} \rangle ::= \text{num} \mid '(' \langle \text{EXPR} \rangle ')'$

Backtracking

- ¿Elijo la primera regla o la segunda?
- 3. * 5

->PROG

->EXPR EOF

$\langle \text{PROG} \rangle ::= \langle \text{EXPR} \rangle \langle \text{PROG} \rangle \mid$
 $\langle \text{EXPR} \rangle \langle \text{EOF} \rangle$

$\langle \text{EXPR} \rangle ::= \langle \text{FACT} \rangle '+' \langle \text{EXPR} \rangle \mid$
 $\langle \text{FACT} \rangle$

$\langle \text{FACT} \rangle ::= \langle \text{FACT} \rangle '*' \langle \text{ATOM} \rangle \mid$
 $\langle \text{ATOM} \rangle$

$\langle \text{ATOM} \rangle ::= \text{num} \mid '(' \langle \text{EXPR} \rangle ')'$

Backtracking

- Si elijo la primera
- 3. * 5

->PROG

->EXPR EOF

->FACT '+' EXPR EOF

$\langle \text{PROG} \rangle ::= \langle \text{EXPR} \rangle \langle \text{PROG} \rangle \mid$
 $\langle \text{EXPR} \rangle \langle \text{EOF} \rangle$

$\langle \text{EXPR} \rangle ::= \langle \text{FACT} \rangle '+' \langle \text{EXPR} \rangle \mid$
 $\langle \text{FACT} \rangle$

$\langle \text{FACT} \rangle ::= \langle \text{FACT} \rangle '*' \langle \text{ATOM} \rangle \mid$
 $\langle \text{ATOM} \rangle$

$\langle \text{ATOM} \rangle ::= \text{num} \mid '(' \langle \text{EXPR} \rangle ')'$

Backtracking

- Si elijo la primera
- El token `*` no coincide, error
- Tengo que hacer backtrack
- `3 * .5`

->PROG

->EXPR EOF

->FACT '+' EXPR EOF

$\langle \text{PROG} \rangle ::= \langle \text{EXPR} \rangle \langle \text{PROG} \rangle \mid$
 $\langle \text{EXPR} \rangle \langle \text{EOF} \rangle$

$\langle \text{EXPR} \rangle ::= \langle \text{FACT} \rangle '+' \langle \text{EXPR} \rangle \mid$
 $\langle \text{FACT} \rangle$

$\langle \text{FACT} \rangle ::= \langle \text{FACT} \rangle '*' \langle \text{ATOM} \rangle \mid$
 $\langle \text{ATOM} \rangle$

Backtracking

- Elijo la otra y sigo
- 3. * 5

->PROG

->EXPR EOF

->FACT '+' <EXPR> EOF----> NO

->FACT EOF

<PROG> ::= <EXPR> <PROG> |
 <EXPR> <EOF>

<EXPR> ::= <FACT> '+' <EXPR> |
 <FACT>

<FACT> ::= <FACT> '*' <ATOM> |
 <ATOM>

<ATOM> ::= num | '(' <EXPR> ')'

Parser descendente recursivo

- Backtracking, desandar, ineficiente
- Tengo que aplicar varias reglas de producción que luego no uso
- Mejor cambiar la gramática
- En casa paso hay un terminal que me deja decidir (lookahead)

Parser descendente recursivo

- Otro problema,
- No puede ser recursivo a la izquierda (ver FACT)
- Para left-most, es un problema, bucle de recursividad infinita
- Luego con la implementación, más evidente

$$\begin{aligned}\langle \text{PROG} \rangle &::= \langle \text{EXPR} \rangle \langle \text{PROG} \rangle \mid \\ &\quad \langle \text{EXPR} \rangle \langle \text{EOF} \rangle\end{aligned}$$
$$\begin{aligned}\langle \text{EXPR} \rangle &::= \langle \text{FACT} \rangle '+' \langle \text{EXPR} \rangle \mid \\ &\quad \langle \text{FACT} \rangle\end{aligned}$$
$$\begin{aligned}\langle \text{FACT} \rangle &::= \langle \text{FACT} \rangle '*' \langle \text{ATOM} \rangle \mid \\ &\quad \langle \text{ATOM} \rangle\end{aligned}$$
$$\langle \text{ATOM} \rangle ::= \text{num} \mid '(' \langle \text{EXPR} \rangle ')'$$

Parser descendente recursivo

- Elimino los prefijos comunes (podría usar la misma estrategia para eliminar la recursividad a la izquierda, si la hubiese)
- En este caso, fácil, PROG, más interesante EXPR

$$\langle \text{PROG} \rangle ::= \langle \text{EXPR} \rangle \langle \text{PROG} \rangle \mid \langle \text{EXPR} \rangle \langle \text{EOF} \rangle$$

- Reescribimos factorizando

$$\langle \text{PROG} \rangle ::= \langle \text{EXPR} \rangle \langle \text{OTHER} \rangle \langle \text{EOF} \rangle$$
$$\langle \text{OTHER} \rangle ::= \langle \text{EXPR} \rangle \langle \text{OTHER} \rangle \mid \langle \text{Empty} \rangle$$

Parser descendente recursivo

$$\langle \text{PROG} \rangle ::= \langle \text{EXPR} \rangle \langle \text{PROG} \rangle \mid \langle \text{EXPR} \rangle \langle \text{EOF} \rangle$$

- En lugar de hacerlo de una, podría darle la vuelta

$$\langle \text{PROG} \rangle ::= \langle \text{PROG} \rangle \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \langle \text{EOF} \rangle$$

- Pero, esto no funcionaría, porque podría tener más de un EOF

Parser descendente recursivo

- Queda (he cambiado FACT para que sea recursivo a la derecha)

$\langle \text{PROG} \rangle ::= \langle \text{EXPR} \rangle \langle \text{OTHER} \rangle \langle \text{EOF} \rangle$

$\langle \text{OTHER} \rangle ::= \langle \text{EXPR} \rangle \langle \text{OTHER} \rangle |$
 $\langle \text{Empty} \rangle$

$\langle \text{EXPR} \rangle ::= \langle \text{FACT} \rangle '+' \langle \text{EXPR} \rangle |$
 $\langle \text{FACT} \rangle$

$\langle \text{FACT} \rangle ::= \langle \text{ATOM} \rangle '*' \langle \text{FACT} \rangle |$
 $\langle \text{ATOM} \rangle$

$\langle \text{ATOM} \rangle ::= \text{num} \mid ' (' \langle \text{EXPR} \rangle ') '$

Factorizar

- El problema es que ambas reglas tienen el mismo prefijo
- Extraigo la parte común

$$\langle XX \rangle ::= \langle PP \rangle \text{'r'} \langle YY \rangle | \\ \langle PP \rangle \text{'m'} \langle ZZ \rangle$$

- Factorizado:

$$\langle XX \rangle ::= \langle PP \rangle \langle JJ \rangle$$
$$\langle JJ \rangle ::= \text{'r'} \langle YY \rangle | \\ \text{'m'} \langle ZZ \rangle$$

Factorizar

- Una de ellas puede estar vacía

$$\langle XX \rangle ::= \langle PP \rangle \text{ 'r' } \langle YY \rangle | \\ \langle PP \rangle$$

- Factorizado:

$$\langle XX \rangle ::= \langle PP \rangle \langle JJ \rangle$$
$$\langle JJ \rangle ::= \text{ 'r' } \langle YY \rangle | \\ \langle \text{Empty} \rangle$$

Parser descendente recursivo

- Sigo teniendo el problema del backtracking
- Lo suyo es factorizar
- Seguimos el ejemplo de antes, FACT
- El problema es que ambas reglas tienen el mismo prefijo de tokens (no terminales que se traducen a tokens)

$\langle \text{EXPR} \rangle ::= \langle \text{FACT} \rangle \text{'+' } \langle \text{EXPR} \rangle |$
 $\langle \text{FACT} \rangle$

- Lo factorizo y saco a otra regla

$\langle \text{EXPR} \rangle ::= \langle \text{FACT} \rangle \langle \text{ADD} \rangle$
 $\langle \text{ADD} \rangle ::= \text{'+' } \langle \text{EXPR} \rangle |$
 $\langle \text{Empty} \rangle$

- Luego sustituyo EXPR por su lado derecho

$\langle \text{EXPR} \rangle ::= \langle \text{FACT} \rangle \langle \text{ADD} \rangle$
 $\langle \text{ADD} \rangle ::= \text{'+' } \langle \text{FACT} \rangle \langle \text{ADD} \rangle |$
 $\langle \text{Empty} \rangle$

Parser descendente recursivo

- Hago esto con toda la gramática y reescribo algunos nombres

$\langle \text{PROG} \rangle ::= \langle \text{EXPR} \rangle \langle \text{OTHER} \rangle \langle \text{EOF} \rangle$

$\langle \text{OTHER} \rangle ::= \langle \text{EXPR} \rangle \langle \text{OTHER} \rangle |$
 $\langle \text{Empty} \rangle$

$\langle \text{EXPR} \rangle ::= \langle \text{TERM} \rangle \langle \text{ADD} \rangle$

$\langle \text{ADD} \rangle ::= '+' \langle \text{TERM} \rangle \langle \text{ADD} \rangle |$
 $\langle \text{Empty} \rangle$

$\langle \text{TERM} \rangle ::= \langle \text{ATOM} \rangle \langle \text{MUL} \rangle$

$\langle \text{MUL} \rangle ::= '*' \langle \text{ATOM} \rangle \langle \text{MUL} \rangle |$
 $\langle \text{Empty} \rangle$

$\langle \text{ATOM} \rangle ::= \text{num} | '(' \langle \text{EXPR} \rangle ')'$

Parser descendente recursivo

- Técnicamente, la regla para OTHER no vale ¿cómo decido entre Empty y EXPR?
- Pero el vacío se detecta EOF así que no es problema

$\langle \text{PROG} \rangle ::= \langle \text{EXPR} \rangle \langle \text{OTHER} \rangle \langle \text{EOF} \rangle$

$\langle \text{OTHER} \rangle ::= \langle \text{EXPR} \rangle \langle \text{OTHER} \rangle |$
 $\langle \text{Empty} \rangle$

- Cambiaría a:

$\langle \text{PROG} \rangle ::= \langle \text{EXPR} \rangle \langle \text{OTHER} \rangle$

$\langle \text{OTHER} \rangle ::= \langle \text{EXPR} \rangle \langle \text{OTHER} \rangle |$
 $\langle \text{EOF} \rangle$

Promoción de terminales

- Si tengo una gramática que no permite decidir con 1 token de lookahead: no es $LL(1)$
- Puedo promocionar terminales de reglas inferiores
- Imaginemos que tengo este fragmento de gramática:

```
<A> ::= <X><A> |  
        'r' <E>  
<X> ::= <Y><Z> |  
        '{' <K> <R> '}',  
<Y> ::= 'p' |  
        'n'
```

- Puedo promocionar (y eliminar) $< Y >$

```
<A> ::= <X><A> |  
        'r' <E>  
<X> ::= 'p' <Z> |  
        'n' <Z> |  
        '{' <K> <R> '}',
```

Promoción de terminales

$$\begin{aligned}\langle A \rangle &::= \langle X \rangle \langle A \rangle \mid \\ &\quad 'r' \langle E \rangle \\ \langle X \rangle &::= 'p' \langle Z \rangle \mid \\ &\quad 'n' \langle Z \rangle \mid \\ &\quad '{' \langle K \rangle \langle R \rangle '}'\end{aligned}$$

- Sigo teniendo $\langle A \rangle$, promociono más.

$$\begin{aligned}\langle A \rangle &::= 'p' \langle X \rangle \langle A \rangle \mid \\ &\quad 'n' \langle X \rangle \langle A \rangle \mid \\ &\quad '{' \langle X \rangle \langle A \rangle \mid \\ &\quad 'r' \langle E \rangle \\ \langle X \rangle &::= \langle Z \rangle \mid \\ &\quad \langle K \rangle \langle R \rangle '}'\end{aligned}$$

- He destapado otros problemas, Luego tengo que seguir con las transformaciones
- Hasta resolver todos los problemas (con cuidado de que tengan sentido los nombres y demás)

Parser descendente recursivo

- Tengo lo que se llama una gramática $LL(1)$
- Left-most, 1 token de lookahead, puedo decidir sin backtracking
- Son un subconjunto de las gramáticas libres de contexto, pero suficientes para casi todo
- Con la factorización me quito la recursividad a la izquierda

Parser descendente recursivo

- Ya puedo implementarlo, es muy sencillo
- Cada regla es una función recursiva

Parser descendente recursivo

```
1 type Parser struct {  
2     l *lex.Lexer  
3     depth int  
4 }  
5 func NewParser(l *lex.Lexer) *Parser {  
6     return &Parser{l, 0}  
7 }
```

Parser descendente recursivo

- Empiezo, por ejemplo

```
1 //EXPR ::= TERM ADD
2 func (p *Parser) Expr() error {
3     if err := p.Term(); err != nil {
4         return err
5     }
6     return p.Add()
7 }
8 //TERM ::= ATOM MUL
9 func (p *Parser) Term() error {
10    if err := p.Atom(); err != nil {
11        return err
12    }
13    return p.Mul()
14 }
```

Parser descendente recursivo

```
1 //MUL ::= '*' ATOM MUL|
2 // Empty
3 func (p *Parser) Mul() error {
4     _, err, isMul := p.match(lex.TokMul)
5     if err != nil || !isMul { //Empty
6         return err
7     }
8     if err := p.Atom(); err != nil {
9         return err
10    }
11
12    return p.Mul()
13 }
```

Parser descendente recursivo

- Me hago una función auxiliar (el token tiene que encajar)

```
1 func (p *Parser) match(tT lex.TokType) (t lex.Token, e error, isMatch bool) {  
2     t, err := p.l.Peek()  
3     if err != nil {  
4         return lex.Token{}, err, false  
5     }  
6     if t.Type != tT {  
7         return t, nil, false  
8     }  
9     t, err = p.l.Lex()  
10    return t, nil, true  
11 }
```

Parser descendente recursivo

```
1 //ATOM ::= num | '(' EXPR ')'  
2 func (p *Parser) Atom() error {  
3     t, err := p.l.Peek()  
4     if err != nil {  
5         return err  
6     }  
7     switch t.Type {  
8     case lex.TokLPar:  
9         t, err := p.l.Lex()  
10        if err := p.Expr(); err != nil {  
11            return err  
12        }  
13        _, err, isRpar := p.match(lex.TokRPar)  
14        if err != nil || !isRpar {  
15            return err  
16        }  
17        case lex.TokNumVal:  
18            t, err := p.l.Lex()  
19            return err  
20        default:  
21            err = errors.New("bad Atom")  
22        }  
23    }  
24 }
```

Parser descendente recursivo

- Para depurar, imprimo el árbol
- Con tabulación (se puede hacer también con paréntesis)

```
1 func (p *Parser) pushTrace(tag string) {  
2     if DebugDesc {  
3         tabs := strings.Repeat("\t", p.depth)  
4         fmt.Fprintf(os.Stderr, "%s%s\n", tabs, tag)  
5     }  
6     p.depth++  
7 }  
8  
9 func (p *Parser) popTrace() {  
10     p.depth—  
11 }
```

Parser descendente recursivo

- Y añado las trazas

```
1 //EXPR ::= TERM ADD
2 func (p *Parser) Expr() error {
3     p.pushTrace(" Expr")
4     defer p.popTrace()
5     if err := p.Term(); err != nil {
6         return err
7     }
8     return p.Add()
9 }
10
11 //TERM ::= ATOM MUL
12 func (p *Parser) Term() error {
13     p.pushTrace(" Term")
14     defer p.popTrace()
15     if err := p.Atom(); err != nil {
16         return err
17     }
18     return p.Mul()
19 }
```

Parser descendente recursivo

- Así para todas y finalmente
- El interfaz será el método **Parse**

```
1 func (p *Parser) Parse() error {  
2     p.pushTrace(" Parse")  
3     defer p.popTrace()  
4     if err := p.Prog(); err != nil {  
5         return err  
6     }  
7  
8     return nil  
9 }
```


Parser descendente recursivo

- Añadimos depuración al lexer también
- Que imprima los tokens en Lex

```
1 func (l *Lexer) Lex() (t Token, err error) {  
2     defer func() {  
3         if DToks {  
4             fmt.Fprintf(os.Stderr, "Lex: %s\n", t)  
5         }  
6     }()  
7     ...  
8 }
```

Parser descendente recursivo

- Y ya podemos probar

...

Expr

Term

Atom

Lex: ['3': TokIntVal[2] 3]

Num['3': TokIntVal[2] 3]

Mul

Lex: ['*': TokMul[42]]

['*': TokMul[42]]

Atom

Lex: ['4': TokIntVal[2] 4]

...

Parser descendente recursivo

- Estamos haciendo el parsing con un tipo de autómata (el programa)
- Con pila (la pila del programa)
- Con esto ya reconocemos lenguajes
- Lo siguiente será añadir acciones
- Para construir el AST o sintetizar atributos

Parser descendente recursivo

- Para tener una gramática adecuada ($LL(1)$, no ambigua, etc.), al final:
 - 1 Empiezo con una ▶ gramática sencilla (ambigua, recursiva a ambos lados etc.)
 - 2 Elimino ambigüedad: ▶ varios niveles o Pratt Parser (siguiente tema)
 - 3 Elimino recursividad a la izquierda, ▶ cambiando de lado , ▶ factorizo los prefijos
 - 4 Si hace falta ▶ cambiando de lado promociono, terminales para que con un token de lookahead se pueda decir qué regla de derivación se aplica.
- La factorización, ayuda a eliminar la recursividad a la izquierda y ayuda a hacer que con un token se decida (evitando varias reglas con el mismo prefijo y por tanto el backtracking). También se puede hacer por la derecha para simplificar la gramática.

Parser descendente recursivo

- Ventajas, implementación sencilla, buenos mensajes de error
- Desventajas: cambios complicados de la gramática, atada al código
- Desventajas: implementación de precedencia en las reglas de derivación, complica la gramática

Manejo de errores

- Imaginemos que tenemos una gramática como ésta
- Estamos en la regla de $\langle ATOM \rangle$ y nos encontramos un '+'

```
<PROG> ::= <EXPR> <PROG> |  
          <EXPR> <EOF>  
<EXPR>  ::= <FACT> '+' <EXPR> |  
          <FACT>  
<FACT>  ::= <FACT> '*' <ATOM> |  
          <ATOM>  
<ATOM>  ::= num | '(' <EXPR> ')'
```

Manejo de errores

```
1 //ATOM ::= num | '(' EXPR ')'  
2 func (p *Parser) Atom() error {  
3     t, err := p.l.Peek()  
4     if err != nil {  
5         return err  
6     }  
7     switch t.Type {  
8         case lex.TokLPar:  
9             t, err := p.l.Lex()  
10             if err := p.Expr(); err != nil {  
11                 return err  
12             }  
13             _, err, isMul := p.match(lex.TokRPar)  
14             if err != nil || !isMul {  
15                 return err  
16             }  
17         case lex.TokNumVal:  
18             t, err := p.l.Lex()  
19             return err  
20         default: //tengo un '+', esperaba un TokLPar o TokNumVal  
21             err = errors.New("bad Atom")  
22     }  
23     return err  
24 }
```

Manejo de errores

- Miro la gramática para ver de dónde vengo

$\langle \text{EXPR} \rangle ::= \langle \text{FACT} \rangle \text{'+' } \langle \text{EXPR} \rangle |$
 $\langle \text{FACT} \rangle$
 $\langle \text{FACT} \rangle ::= \langle \text{FACT} \rangle \text{'*'} \langle \text{ATOM} \rangle |$
 $\langle \text{ATOM} \rangle$
 $\langle \text{ATOM} \rangle ::= \text{num} \mid \text{'(' } \langle \text{EXPR} \rangle \text{' ')}$

```
20 //ATOM ::= num | '(' EXPR ')'  
21     default: //tengo un '+', esperaba un TokLPar o TokNumVal  
22     err = errors.New("bad Atom")  
23 }  
24 return err  
25 }
```


Manejo de errores

- Miro la gramática para ver de dónde vengo

$\langle \text{EXPR} \rangle ::= \langle \text{FACT} \rangle \text{'+' } \langle \text{EXPR} \rangle |$
 $\langle \text{FACT} \rangle$
 $\langle \text{FACT} \rangle ::= \langle \text{FACT} \rangle \text{'*'} \langle \text{ATOM} \rangle |$
 $\langle \text{ATOM} \rangle$
 $\langle \text{ATOM} \rangle ::= \text{num} | \text{'(' } \langle \text{EXPR} \rangle \text{' ')}$

```
20 //ATOM ::= num | '(' EXPR ')'  
21     default: //tengo un '+', esperaba un TokLPar o TokNumVal  
22     err = errors.New("bad Atom")  
23 }  
24 return err  
25 }
```

Manejo de errores

- Quiero seguir haciendo parsing
- Para poder dar más errores
- Tengo que volver a un punto razonable (resincronizar)
- Tengo tres estrategias:
 - ▶ Inventarme lo que esperaba (paréntesis o número), es decir, insertar un token
 - ▶ Buscar acabar una construcción (buscar un token de sincronización)
 - ▶ Devolver un error y que se sincronice más arriba
- El programa de entrada ya es incorrecto, busco poder dar otros errores, no voy a generar código
- Haga lo que haga siempre hay una entrada que lo rompe y hay errores en cascada → número máximo de errores

Manejo de errores: resincronización

- Si es evidente, se puede inyectar un token (no se puede hacer mucho)
- Si no, se suelen buscar caracteres que terminen la construcción en este nivel (por ejemplo ';' en una sentencia, ')') o un operador en una expresión... y se tira la entrada hasta ese operador (incluido o no, depende)
- Puede tener una función `lexUntilOneOf` o algo parecido para ayudar (en el lexer)
- Es un arte y un equilibrio delicado ver qué se hace, depende de la construcción: evitar estrategias muy complicadas

Manejo de errores

- En este caso concreto, seguramente insertaría un num (o retornaría, que es equivalente)
- Considero que he arreglado el error, retorno con éxito (pero me apunto que ha habido un error)

```
<EXPR> ::= <FACT> '+' <EXPR> |  
          <FACT>  
<FACT>  ::= <FACT> '*' <ATOM> |  
          <ATOM>  
<ATOM>  ::= num | '(' <EXPR> ')'
```

```
20 //ATOM ::= num | '(' EXPR ')'  
21     default: //tengo un '+', esperaba un TokLPar o TokNumVal  
22     err = errors.New("bad Atom")  
23 }  
24 return err  
25 }
```

Manejo de errores

```
20 //ATOM ::= num | '(' EXPR ')'  
21 func (p *Parser) Atom() error {  
22     t, err := p.l.Peek()  
23     if err != nil {  
24         return err  
25     }  
26     switch t.Type {  
27     case lex.TokLPar:  
28         t, err := p.l.Lex()  
29         if err := p.Expr(); err != nil {  
30             return err  
31         }  
32         _, err, isMul := p.match(lex.TokRPar)  
33         if err != nil || !isMul {  
34             return err  
35         }  
36     case lex.TokNumVal:  
37         t, err := p.l.Lex()  
38         return err  
39     default:  
40         p.Errorf("bad Atom, expected ( or number, got %s", t.Type)  
41         //Error noted, I will act as if nothing happened  
42         // alternatively, I could look for lex.ToxRPar  
43         err = nil  
44     }  
45     return err  
46 }
```

Manejo de errores

```
20 func (p *Parser) Errorf(s string, v ...interface{}) {
21     fmt.Fprintf(os.Stderr, s+"\n", v...)
22     p.nErr++ //no code will be generated
23     if p.nErr >= maxErrors {
24         log.Fatalf("too many errors")
25     }
26 }
```

Manejo de errores: AST

- Se insertan nodos erróneos (tipo de símbolo error) para ignorar más adelante en comprobaciones semánticas
- Cuando se comprueben tipos, etc. se ignoran y no se da errores sobre ellos

Parser descendente recursivo: precedencia

- Muy complicado si hay muchos niveles de precedencia
- Solución: “Pratt parsing” o “Precedence climbing”, o “Shunting-yard algorithm” variantes de la misma idea
- Cada nivel de precedencia tiene un número asociado que es el “binding power”, poder para atar
- Los subárboles se pegan dependiendo de este número

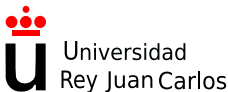
Tema 3.1: Parsing descendente de expresiones

Pratt parsing

Gorka Guardiola Múzquiz

GSYC

23 de noviembre de 2022



Operadores con precedencia

- Ya hemos visto que las gramáticas tienen ambigüedad
- Y la deambiguación la hace la precedencia
- Codificar esto en la gramática, complicado
- Cuatro niveles de precedencia, operadores a derechas e izquierdas. . .

Operadores con precedencia

- **Pratt Parsing (también llamado top down operator precedence parser)**

Pratt, Vaughan. "Top down operator precedence." Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (1973).

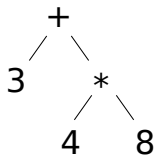
- **Precedence Climbing** Richards, Martin; Whitby-Stevens, Colin (1979). BCPL — the language and its compiler. Cambridge University Press. ISBN 9780521219655.

- **Shunting-yard Algorithm** Dijkstra, E.W. (1961). Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60. Stichting Mathematisch Centrum. Rekenafdeling. Stichting Mathematisch Centrum.

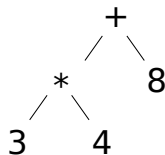
- Esencialmente lo mismo, el Shunting-yard utiliza una pila separada (explícita, no la del programa) y hay detalles de implementación, pero la idea es la misma. Vamos a ver sólo el Pratt parser.

AST, precedencia

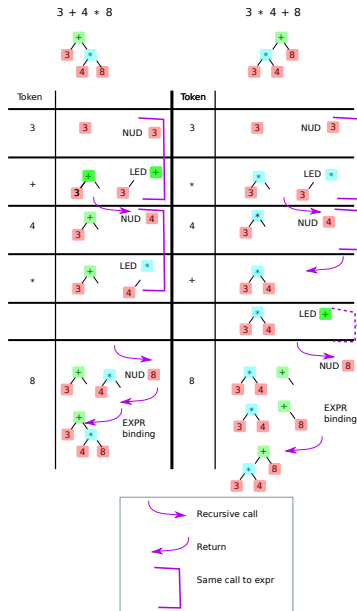
$3 + 4 * 8$



$3 * 4 + 8$



AST, construcción



Pratt Parser: ideas

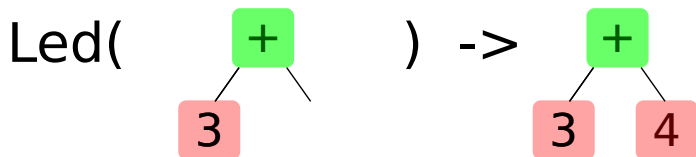
- Tres funciones, Nud, Led, Expr

Nud : Null-denotation: (desnuda, juego de palabra), son átomos (expresiones entre paréntesis, símbolos terminales, etc.)

Led : Left-denotation: con contexto a la izquierda, llama a Expr recursivamente

Expr : Construcción de la expresión, llamo una vez a Nud y voy llamando a Led mientras me deje la precedencia

Nud() -> 3



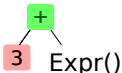
Expr

$3 + 4 * 8$

Expr()

Nud() -> 3

Led(3 , +) ->



```
graph TD; A[+] --- B[3]; A --- C[Expr()];
```

Nud() -> 4

Led(4 , *) ->



```
graph TD; A[*] --- B[4]; A --- C[Expr()];
```

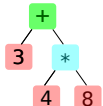
Nud() -> 8

->



```
graph TD; A[*] --- B[4]; A --- C[8];
```

->



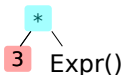
```
graph TD; A[+] --- B[3]; A --- C[*]; C --- D[4]; C --- E[8];
```

$$3 * 4 + 8$$

Expr()

Nud() -> 3

Led(3 , *) ->



```

graph TD
    A["*"] --- B["3"]
    A --- C["Expr()"]
  
```

Nud() -> 4

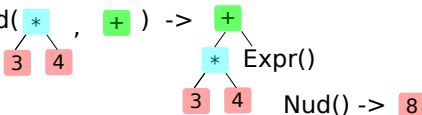
->



```

graph TD
    A["*"] --- B["3"]
    A --- C["4"]
  
```

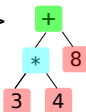
Led(* , +) ->



```

graph TD
    A["+"] --- B["*"]
    A --- C["Expr()"]
    B --- D["3"]
    B --- E["4"]
    C --- F["Nud() -> 8"]
  
```

->



```

graph TD
    A["+"] --- B["*"]
    A --- C["8"]
    B --- D["3"]
    B --- E["4"]
  
```

Tabla de precedencia

- Ojo, aquí los paréntesis son operadores

```
1 var precTab = map[rune] int {
2     ')': 1,
3     '+': 20,
4     '-': 20,
5     '*': 30,
6     '^': 40,
7     '(' : 50,
8 }
9 const defRbp = 0
10
11 func bindPow(tok lex.Token) int {
12     if rbp, ok := precTab[rune(tok.Type)]; ok {
13         return rbp
14     }
15     return defRbp
16 }
```

Tablas

- En el original, cada uno tiene su Nud y su Led
- Esto es una simplificación

```
1 var leftTab = map[rune] bool{  
2     '^': true ,  
3 }  
4 var unaryTab = map[rune] bool{  
5     '+': true ,  
6     '-': true ,  
7     '(': true ,  
8 }
```

- Null-denotation (sin contexto por la izquierda)

```
1 func (p *Parser) Nud(tok lex.Token) (expr *Expr, err error) {
2     var rExpr *Expr
3     var rbp int
4     p.dPrintf("Nud: %d, %s \n", rbp, tok)
5     if tok.Type == lex.TokLPar { //special unary, parenthesis
6         expr, err = p.Expr(rbp)
7         if err != nil {
8             return nil, err
9         }
10        if _, err, isClosed := p.match(lex.TokRPar); err != nil {
11            return nil, err
12        } else if !isClosed {
13            return nil, errors.New("unmatched parenthesis")
14        }
15        return expr, nil
16    }
```

- Null-denotation (sin contexto por la izquierda)

```
17  expr = NewExpr(tok)
18  rbp = bindPow(tok)
19  rTok := rune(tok.Type)
20  if rbp != defRbp { //regular unary operators
21      if !unaryTab[rTok] {
22          errs := fmt.Sprintf("%s is not unary", tok.Type)
23          return nil, errors.New(errs)
24      }
25      rExpr, err = p.Expr(rbp)
26      if rExpr == nil {
27          return nil, errors.New("unary operator: no operand")
28      }
29      expr.ERight = rExpr
30  }
31  return expr, nil
32 }
```

- Left-denotation (contexto por la izquierda)

```
1 func (p *Parser) Led(left *Expr, tok lex.Token) (expr *Expr, err error) {
2     var rbp int
3     expr = NewExpr(tok)
4     expr.ELeft = left
5     rbp = bindPow(tok)
6     if isleft := leftTab[rune(tok.Type)]; isleft {
7         rbp -= 1
8     }
9     p.dPrintf("Led: %d, {{%s}} %s \n", rbp, left, tok)
10    rExpr, err := p.Expr(rbp)
11    if err != nil {
12        return nil, err
13    }
14    if rExpr == nil {
15        errs := fmt.Sprintf("missing operand for %s\n", tok.Type)
16        return nil, errors.New(errs)
17    }
18    expr.ERight = rExpr
19    return expr, nil
20 }
```

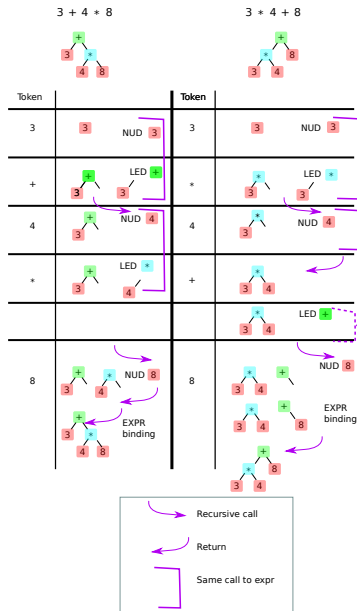
- Esta es la función del descendente para reconocer una expresión

```
1 func (p *Parser) Expr(rbp int) (expr *Expr, err error) {
2     var left *Expr
3     s := fmt.Sprintf(" Expr: %d", rbp)
4     p.pushTrace(s)
5     defer p.popTrace(&err)
6
7     tok, err := p.l.Peek()
8     if err != nil {
9         return expr, err
10    }
11    p.dPrintf("expr: Nud Lex: %s", tok)
12    if tok.Type == lex.TokEof {
13        return expr, nil
14    }
15    p.l.Lex() //already peeked
16    if left, err = p.Nud(tok); err != nil {
17        return nil, err
18    }
19    expr = left
```



```
20  for {
21      tok, err := p.l.Peek()
22      if err != nil {
23          return expr, err
24      }
25      if tok.Type == lex.TokEof || tok.Type == lex.TokRPar {
26          return expr, nil
27      }
28      if bindPow(tok) <= rbp {
29          p.dPrintf("Not enough binding: %d <= %d, %s\n",
30              bindPow(tok), rbp, tok)
31          return left, nil
32      }
33      p.l.Lex() //already peeked
34      p.dPrintf("expr: led Lex: %s", tok)
35      if left, err = p.Led(left, tok); err != nil {
36          return expr, err
37      }
38      expr = left
39  }
40  return expr, err
```

AST, Pratt



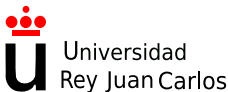
Tema 4: Acciones y atributos

Autómatas y desarrollo avanzado de software

Gorka Guardiola Múzquiz

GSYC

23 de noviembre de 2022



Una derivación

- Recorre un árbol (puede o no construirlo)
- Dependiendo de dónde ponga las llamadas
- Va a recorrer el árbol de una manera

Formas de recorrer un árbol

- En profundidad (DFS, depth first): Preorden, postorden, inorden (*Pre-order, post-order, in-order*)
- En anchura (BFS, breadth first)

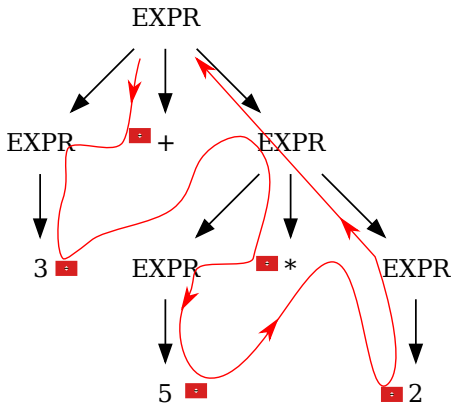
Ejemplo de árbol

- Ejemplo de árbol

```
1 const (  
2     Right = iota  
3     Left  
4     NSides  
5 )  
6  
7 type Tree struct {  
8     c rune  
9     isLeaf bool  
10    child [NSides]* Tree  
11 }
```

Formas de recorrer un árbol

- DFS, preorden
- $3 + 5 * 2 \rightarrow + 3 * 5 2$
- Coincide por casualidad con el anterior
- Notación polaca, útil para evaluar (como si los operadores fueran funciones)

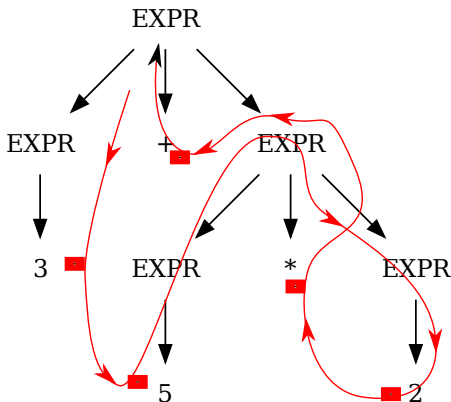


Preorden

```
1
2 func (t *Tree) String() (s string) {
3     if t.isLeaf {
4         return fmt.Sprintf("[%c]", t.c)
5     }
6     s += fmt.Sprintf("[%c]", t.c)
7     s += t.child[Left].String()
8     s += t.child[Right].String()
9     return s
10 }
```


Formas de recorrer un árbol

- DFS, postorden
- $3 + 5 * 2 \rightarrow 3 \ 5 \ 2 \ * \ +$
- Notación polaca inversa, útil para calculadora, metes operandos y acción

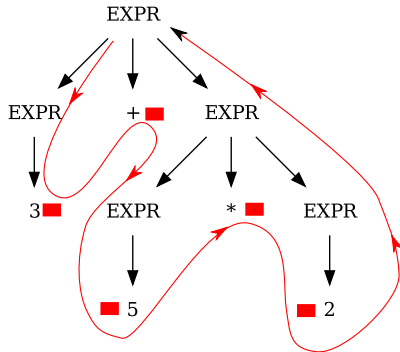


Postorden

```
1
2 func (t *Tree) String() (s string) {
3     if t.isLeaf {
4         return fmt.Sprintf("[%c]", t.c)
5     }
6     s += t.child[Left].String()
7     s += t.child[Right].String()
8     s += fmt.Sprintf("[%c]", t.c)
9     return s
10 }
```

Formas de recorrer un árbol

- DFS, inorden
- $3 + 5 * 2 \rightarrow 3 + 5 * 2$
- Para reescribir el código tal cual

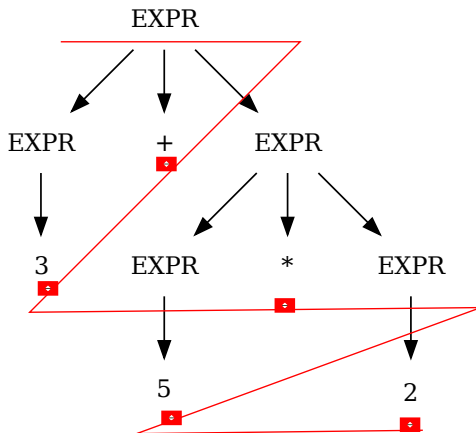


inorden

```
1 func (t *Tree) String() (s string) {  
2     if t.isLeaf {  
3         return fmt.Sprintf("[%c]", t.c)  
4     }  
5     s += t.child[Left].String()  
6     s += fmt.Sprintf("[%c]", t.c)  
7     s += t.child[Right].String()  
8     return s  
9 }
```

Formas de recorrer un árbol

- En anchura (BFS, breadth first)
- Por niveles
- $3 + 5 * 2 \rightarrow +3 * 5 2$
- No tiene mucho sentido para una expresión



BFS

```
10 func (t *Tree) StrDepth(depth int, depthWant int) (s string) {
11     if depth == depthWant {
12         s += fmt.Sprintf("%c", t.c)
13     } else if !t.isLeaf {
14         depth++
15         s += t.child[Left].StrDepth(depth, depthWant)
16         s += t.child[Right].StrDepth(depth, depthWant)
17     }
18
19     return s
20 }
21
22 func (t *Tree) Bfs() (s string) {
23     depth := t.Depth(0)
24     fmt.Println(depth)
25     for i := 0; i < depth; i++ {
26         s += t.StrDepth(0, i)
27     }
28     return s
29 }
```

BFS

```
30 func Max(a, b int) int {
31     if a > b {
32         return a
33     } else {
34         return b
35     }
36 }
37
38 func (t *Tree) Depth(depth int) (d int) {
39     depth++
40     if t.isLeaf {
41         return depth
42     }
43     d1 := t.child[Left].Depth(depth)
44     d2 := t.child[Right].Depth(depth)
45     return Max(d1, d2)
46 }
```

Atributos

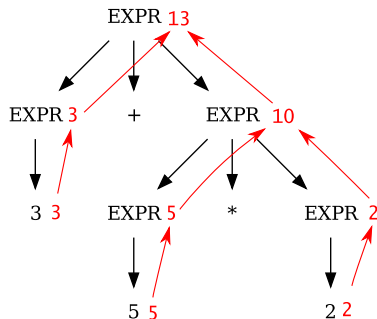
- Asociar un valor a un nodo
- Sintéticos y heredados
- Los sintéticos dependen del subárbol (los hijos)
- Los heredados dependen del superárbol (se hereda de los antepasados)

Acciones

- Asociar un efecto lateral (imprimir algo, por ejemplo)
- A un nodo del AST
- Cuando lo recorro (al aplicar una derivación, o recorriéndolo tras construirlo)
- Es importante cómo lo recorro

Ejemplo de la calculadora

- Evaluar el árbol sin construirlo
- Se evalúan preorden, y se guardan en atributos sintéticos
- Cada subexpresión (nodo del árbol) tiene asociado un atributo, un valor



Ejemplo de la calculadora

- Todo como antes, pero cada función (nodo) devuelve un valor
- Esta versión más sencilla, sólo permite una expresión por programa

```
1 //PROG ::= EXPR[val] EOF
2 func (p *Parser) Prog() error {
3     p.pushTrace("Prog")
4     defer p.popTrace()
5     val, err := p.Expr();
6     if err != nil {
7         return err
8     }
9     t, err, isEof := p.match(lex.TokEof)
10    if err != nil {
11        return err
12    }
13    if !isEof {
14        es := fmt.Sprintf("need %s, got %s", lex.TokEof, t.Type)
15        return errors.New(es)
16    }
17    fmt.Printf("The result is %v\n", val)
18    return nil
19 }
```

Ejemplo de la calculadora

- Y así con todos

```
20 //EXPR[val] ::= TERM[val] ADD[val]
21 func (p *Parser) Expr() (float64, error) {
22     p.pushTrace("Expr")
23     defer p.popTrace()
24     val1, err := p.Term()
25     if err != nil {
26         return 0, err
27     }
28     val2, err := p.Add()
29     return val1 + val2, err
30 }
```

Ejemplo de la calculadora

- Y así con todos

```
31 //TERM[val] := ATOM[val] MUL[val]
32 func (p *Parser) Term() (float64, error) {
33     p.pushTrace("Term")
34     defer p.popTrace()
35     val1, err := p.Atom()
36     if err != nil {
37         return 0, err
38     }
39     val2, err := p.Mul()
40     return val1 * val2, err
41 }
```

Ejemplo de la calculadora

- Y así con todos

```
42 //ATOM[val] ::= num | '(' EXPR ')'  
43 func (p *Parser) Atom() (float64, error) {  
44     p.pushTrace("Atom")  
45     defer p.popTrace()  
46     t, err := p.l.Peek()  
47     if err != nil {  
48         return 0, err  
49     }
```

Ejemplo de la calculadora

```
50  var val float64
51  switch t.Type {
52      case lex.TokLPar:
53          t, err := p.l.Lex()
54          val, err = p.Expr();
55          if err != nil {
56              return 0, err
57          }
58          p.pushTrace("Lpar"+t.String())
59          defer p.popTrace()
60          _, err, isMul := p.match(lex.TokRPar)
61          if err != nil || !isMul{
62              return 0, err
63          }
```

Ejemplo de la calculadora

```
51     case lex.TokFloatVal:
52         t, err := p.l.Lex()
53         if err != nil {
54             return 0, err
55         }
56         val = t.TokFloatVal
57         p.pushTrace("Num"+t.String())
58         defer p.popTrace()
59     default:
60         err = errors.New("bad Atom")
61         return 0, err
62 }
63 return val, err
64 }
```


Ejemplo de la calculadora

- Construir el árbol y luego evaluarlo
- No confundir tokens, símbolos, e identificadores: lex, AST, declaraciones, más adelante
- Idea sencilla de árbol

Construir el árbol y luego evaluarlo

Idea sencilla de árbol (con símbolos)

```
51 type Sym struct {  
52     name      string //name of the symbolic constants  
53     sType     int     //SKey, SConst, SExpr...  
54  
55     lex.Place //file and line no  
56     //other stuff for the tree  
57  
58     /* one of */  
59     FloatVal float64  
60     Expr    *Expr  
61     val     *Sym /* for constants/literals */  
62 }
```

Construir el árbol y luego evaluarlo

AST

```
51 type Expr struct {  
52     Left *Sym  
53     Right *Sym  
54     Op int  
55     Val *Sym  
56 }
```

Ejemplo de AST más completo

- Imaginemos que se añaden acciones al parser para construir el AST (y luego generar código. . .)
- El AST está construido a base de símbolos, terminales y no terminales de la gramática (la original, limpia, antes de convertirla en $LL(1)$ o hacer otras transformaciones)
- Ojo, como siempre, elegir nombres descriptivos para la gramática
- El árbol, técnicamente va a estar compuesto de símbolos, pero hay varias alternativas de implementación, dependiendo del lenguaje

Símbolos

- Va a haber un tipo de símbolo por elemento de la gramática
- Los símbolos irán en un paquete separado (veremos más adelante más detalles)
- Por ejemplo, para un lenguaje de programación imperativo con un sólo programa que representa una función

```
51 const (  
52     SNone    = iota  
53     SKey     // keyword  
54     SStr     // a string buffer  
55     SConst   // constant or literal  
56     SType    // type definition  
57     SVar     // object definition  
58     SUnary   // unary expression  
59     SBinary  // binary expression  
60     SProc    // procedure  
61     SFunc    // function  
62     SFCall   // procedure or function call  
63 )
```

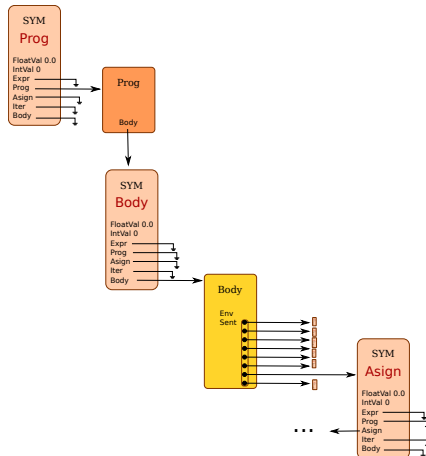
Símbolos

- Se podrían poner los campos directamente en el símbolo, pero es más limpio tener Iter y demás para organizarlo y que ocupe menos

```
51 type Sym struct {  
52     name      string //name of the var, literal, etc.  
53     sType     int    //SKey, SConst...  
54     DataType *Type  //for later, when we check types  
55     lex.Place //file and line no  
56     //other stuff for the tree  
57  
58     /* one of */  
59     FloatVal float64  
60     IntVal  int64  
61     Expr   *Expr  
62     Prog   *Prog  
63     Body   *Body  
64     Asign  *Asign  
65     Iter   *Iter  
66     val    *Sym /* for var when eval */  
67 }
```

Símbolos

- El tipo dice qué punteros apuntan a algo



Símbolos

- Alternativamente, se podría hacer también así
- Luego habrá que hacer métodos que devuelvan el contenido después de hacer un cast/type assertion/type switch

```
51 type Sym struct {  
52     name      string //name of the var, literal, etc.  
53     sType     int    //SKey, SConst...  
54  
55     DataType *Type //for later, when we check types  
56     lex.Place   //file and line no  
57     //other stuff for the tree  
58  
59     symContent interface {}  
60 }
```


AST sencillo

```
51 type Expr struct {  
52     Left *Sym  
53     Right *Sym  
54     Op int  
55     Val *Sym  
56 }  
57 type Prog struct {  
58     b *Body  
59 }  
60 type Asign struct {  
61     LVal *Sym  
62     RVal *Sym  
63 }
```

AST sencillo

```
51 type Iter struct {  
52     St *Sym  
53     End *Sym  
54     b *Body  
55 }  
56 type Body struct {  
57     lvl int      /* for printing */  
58     env Env  
59     sent []*Sym /* iter expr asig */  
60 }
```

Símbolos

- El símbolo podría ser un interfaz que cumplen `*Prog`, `*Iter` y demás, hay más alternativas
- `*Prog`, `*Iter` y demás, podrían ser también interfaces
- En lenguajes con tipos variantes o uniones (*sum type*), se puede hacer un poco diferente

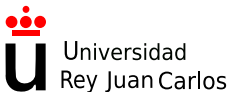
Tema 5: Símbolos y ámbitos

Autómatas y desarrollo avanzado de software

Gorka Guardiola Múzquiz

GSYC

23 de noviembre de 2022



En un programa se declaran identificadores

- Imports, tipos, constantes, funciones, variables

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 type Z int
8 const scaleFact = 3
9
10 func (a Z) scale() Z {
11     s := a
12     return s * scaleFact
13 }
14
15 func main() {
16     fmt.Println(Z(3).scale())
17 }
```

Ámbitos

- Tiene que haber una declaración por cada n símbolos
- Recordar el autómata con pila 0^n1^n , (lenguaje **libre de contexto o tipo 2**)
- Para hacer más sofisticado para declaraciones, es similar a $0^n1^m2^n3^m$, ya **dependiente de contexto o tipo 1**
- Se aumenta el compilador con una pila de ámbitos (scopes), ya se tienen dos pilas, la del programa y la de ámbitos que **permite uno de tipo 0 o enumerable recursivo** es una máquina de Turing
- Recordar: **Ámbito (scope)**: *región del código en el que es válido un identificador (tiempo de compilación)*

Ámbitos

- Jerarquía: Global, paquete, fichero, función, bloque
- Depende del lenguaje
- Orden: importa el orden de declaraciones o no

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 type Z int
8 const scaleFact = 3
9
10 func (a Z) scale() Z {
11     s := a
12     return s * scaleFact
13 }
14
15 func main() {
16     fmt.Println(Z(3).scale())
17 }
```

Ámbitos en compilación

- En **tiempo de compilación**, ámbitos léxicos
- Son una pila, cuando estoy haciendo el parsing entrar es un push, salir es un pop
- Hay uno global para builtins y primitivas del lenguaje (tipos incluidos)
- Se usa para comprobar declaraciones y tipos en compilación
- Generar tablas de símbolos para el enlazado

Ámbitos en ejecución

- Tengo que tener en cuenta no tanto los nombres como la asignación de memoria/recursos, ciclo de vida
- Variables locales, parámetros, instanciación de paquetes. . .
- Se usa la pila del programa **en ejecución** (o no, análisis de escape, destructores)
- No confundir con compilación
- El compilador tendrá que **generar código** que lo haga

Símbolos vs tokens

- Lo que pinchamos en el árbol son símbolos (de la gramática), lo que devuelve el lexer son tokens
- Los símbolos, pueden tener nombre, tipo (de símbolo y de datos)...
 - ▶ Tokens, tipo de token (función como elemento sintáctico) → Símbolos terminales, tipo de símbolo
 - ▶ Símbolos → Símbolos no terminales, tipo de símbolo
- Se crean en el ámbito adecuado, incluso para constantes, keywords (como identificadores en ámbito global si hiciese falta)
- Permiten que el lenguaje y el usuario usen los mismos mecanismos (limpio), keywords como identificadores o tipos predefinidos, por ejemplo
- A veces los literales llevan su propio símbolo, como constantes anónimas (strings, enteros): permite optimizaciones

Tabla de símbolos

- Contendrá los símbolos asociados a nombres (identificadores)
- Para construir un parser hay que declarar los símbolos para el AST y la tabla de símbolos para los ámbitos
- Ojo, símbolos e identificadores no son lo mismo, unos van a la tabla de símbolos y otros son sencillamente nodos del AST (los identificadores son símbolos, pero no todos los símbolos son identificadores)

Símbolos

- Tipos de símbolos

```
1 const (  
2     SNone    = iota  
3     SKey     // keyword  
4     SStr     // a string buffer  
5     SConst   // constant or literal  
6     SType    // type definition  
7     SVar     // object definition  
8     SUnary   // unary expression  
9     SBinary  // binary expression  
10    SProc    // procedure  
11    SFunc    // function  
12    SFCall   // procedure or function call  
13 )
```

Símbolos

- Tipos de símbolos
- Muchos han promocionado desde un token (símbolos terminales)
- Se puede alternatively guardar el propio token en el símbolo (si tiene sentido)

```
1 type Sym struct {  
2     name    string    //name of the var, literal, etc.  
3     sType   int       //SKey, SConst...  
4     tType   *Type     //type for later  
5  
6     tokKind int       //For the grammar  
7                       //OJO a las dependencias circulares TokType.  
8     FloatVal float64  //may be taken from the token  
9     /*...more */  
10  
11     fName   string    //to locate declaration, from token  
12     lineNo  int  
13     //other stuff for the tree  
14 }
```

Símbolos

- Cada ámbito, una tabla hash, clave nombre (a veces otros atributos) del símbolo
- En Go, por ejemplo, un map
- Así que vamos a tener una pila de tablas hash

Símbolos

- Es interesante tener la tabla de símbolos en un paquete separado
- Compartido y usado entre scanner y parser
- Ojo, **Go no permite dependencias circulares entre paquetes** (ciclos en el grafo de imports)

```
1 type Env map[string]*Sym
2
3 type StkEnv []Env
4
5 func (envs *StkEnv) PushEnv() {
6     env := Env{}
7     *envs = append(*envs, env)
8 }
9 func (envs *StkEnv) PopEnv() {
10    eS := *envs
11    if len(eS) == 1 {
12        panic("cannot pop builtin")
13    }
14    *envs = eS[:len(eS)-1]
15 }
```

- Se pueden añadir otros parámetros (fName, lineNo. . .)

```
1 func (envs *StkEnv) NewSym(name string, sType int)(s *Sym, err error) {  
2     eS := *envs  
3     s = &Sym{sType: sType}  
4     e := eS[len(eS)-1]  
5     if _, ok := e[name]; ok {  
6         //permit shadowing...  
7         return nil, errors.New("already declared sym")  
8     }  
9     e[name] = s  
10    return s, nil  
11 }
```


Símbolos

- En el ámbito global predefinidos:

```
1  var s sym.StkEnv
2  s.PushEnv() //global (builtins)
3  s.defKeyWords()
4  s.defTypes()
5  ...
6  s.PushEnv() //from now on
```

Símbolos

- Por ejemplo, keywords
- En go no se pueden hacer mapas constantes o inmutables (este lo sería si se pudiese)
- Sabiendo todos los elementos, se podría usar una hash perfecta

```
1 var sKeyTab = map[string]int{
2     "False":    FALSE,
3     "True":     TRUE,
4     "and":      AND,
5     "array":    ARRAY,
6     "case":     CASE,
7     ...
8     "for":      FOR,
9     "function": FUNCTION,
10    "if":       IF,
11    ...
12    "not":      NOT,
13    "of":       OF,
14    "or":       OR,
15    "procedure": PROCEDURE,
16    ...
17    "vars":     VARS,
18    "while":    WHILE,
19 }
```

Símbolos

- En el ámbito global predefinidos
- Constantes, variables, procedimientos, funciones, tipos, literales...

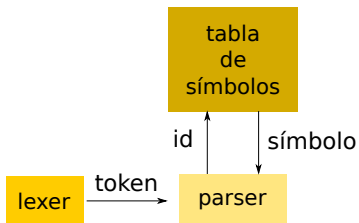
```
1 for v, k := range(sKeyTab) {  
2     s, err := stkEnv.NewSym(k, Skey)  
3     if err != nil {  
4         return err  
5     }  
6     s.tokKind = v  
7     s.fName = "builtin"  
8 }
```

The lexer hack

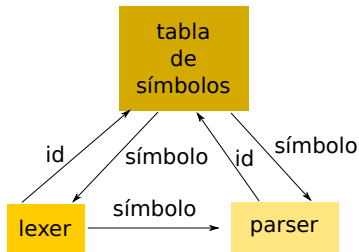
- El lexer mira en la tabla de símbolos
- Mejor que devolver siempre ID, o sólo resolver keywords
- El lexer mira la tabla de símbolos y devuelve símbolos
- Devuelve TYPEID o VARID o FUNCID (predefinidos o no)
- Cada uno su tabla, en un orden ("Bool" $\rightarrow$ ID $\rightarrow$ {TYPE, BOOL})
- Simplifica la gramática
- https://en.wikipedia.org/wiki/The_lexer_hack

The lexer hack

Sin Lexer Hack



Con Lexer Hack



Resolver un símbolo

- Depende de las reglas de shadowing del lenguaje, si está prohibido o no
- Voy del ámbito más interior hacia fuera

```
1 func (envs *StkEnv) GetSym(name string) (s *Sym) {  
2     eS := *envs  
3     for i := len(eS)-1; i >= 0; i— {  
4         if s, ok := eS[i][name]; ok {  
5             return s  
6         }  
7     }  
8     return nil  
9 }
```

Ejemplo

- Añadir constantes al descendente

```
1 type Parser struct {  
2     l *lex.Lexer  
3     depth int  
4     stkEnv sym.StkEnv  
5 }  
6  
7 type Const struct {  
8     cType lex.TokType  
9     val float64  
10 }  
11  
12 var sConst = map[string] Const{  
13     "Pi":      Const{lex.TokFloatVal, math.Pi},  
14 }
```

Ejemplo

- Añadir constantes al descendente

```
1
2 func (p *Parser) initSyms() error {
3     for k, v := range(sConst) {
4         s, err := p.stkEnv.NewSym(k, sym.SConst)
5         if err != nil {
6             return err
7         }
8         s.TokKind = int(v.cType)    //TODO circ dependencies, this is done with type
9         s.FloatVal = v.val
10        s.FName = "builtin"
11    }
12    return nil
13 }
```


Ejemplo

- Añadir constantes al descendente

```
1 func NewParser(l *lex.Lexer) *Parser {  
2     p := &Parser{l, 0, nil}  
3     p.stkEnv.PushEnv() //global  
4     p.initSyms()  
5     p.stkEnv.PushEnv()  
6     return p  
7 }
```

Ejemplo, añadir constantes

- En Atom

```
1 func (p *Parser) Atom() (float64, error) {  
2   ...  
3   case lex.TokId:  
4     t, err := p.l.Lex()  
5     if err != nil {  
6       return 0, err  
7     }  
8     p.pushTrace("ID"+t.String())  
9     defer p.popTrace()  
10    s := p.stkEnv.GetSym(t.Lexema)  
11    if s == nil {  
12      return -1, errors.New("ID not found: " + t.Lexema)  
13    }
```

Ejemplo

- En Atom

```
14         if lex.TokFloatVal != lex.TokType(s.TokKind) {
15             return -1, errors.New("ID bad wrong sym kind")
16         }
17         val = s.FloatVal
18         p.pushTrace("const" + t.String())
19         defer p.popTrace()
20     ...
21 }
```

Ejemplo

- Si es muy sencillo, como este caso
- Evitar el Lexer Hack (más modular, paralelizable)

Añadir variables

- Al principio del ámbito hago push y al final pop

```
1 //BODY ::= '{' SENTS '}'
2 func (p *Parser) Body() (lines *Lines, e error) {
3     p.pushTrace("Body")
4     defer p.popTrace(&e)
5     tok, err, isBr := p.match(lex.TokLBrack)
6     if err != nil || !isBr {
7         return nil, errors.New("Body expected {, found:"+tok.String())
8     }
9     p.pushTrace("LBrack" + tok.String())
10    defer p.popTrace(nil)
11    p.stkEnv.PushEnv()
12    defer p.stkEnv.PopEnv()
13    if lines, err = p.Lines(); err != nil {
14        return lines, err
15    }
16    tok, err, isBr := p.match(lex.TokRBrack)
17    if err != nil || !isBr {
18        return lines, err
19    }
20    p.pushTrace("RBrack" + tok.String())
21    defer p.popTrace(nil)
22
23    return expr, err
24 }
```

Añadir variables

- Y luego necesito la construcción para declaraciones

```
1 //DECL ::= 'var' ID
2 func (p *Parser) Decl() (dec *Decl, e error) {
3     p.pushTrace("Decl")
4     defer p.popTrace(&e)
5     _, err, isVar := p.match(lex.TokVar)
6     if err != nil || !isVar{
7         return nil, err
8     }
9     tok, err, isvar := p.match(lex.TokId)
10    if err != nil {
11        return nil, errors.New("Decl, expected ID found:" + tok.String())
12    }
13    tok, err = p.l.Lex()
14    if err != nil {
15        return nil, err
16    }
17    p.pushTrace("ID" + tok.String())
18    defer p.popTrace()
19    s := p.stkEnv.NewSym(tok.Lexema, Svar)
20    if s != nil {
21        return nil, errors.New("ID already defined: " + tok.Lexema)
22    }
23    s.TokKind = int(lex.TokFloatVal)
24    p.pushTrace("var" + tokString())
25    defer p.popTrace()
26
27    return s, err
28 }
```

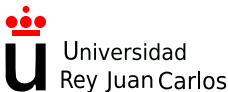
Tema 6: Parsing ascendente

Autómatas y desarrollo avanzado de software

Gorka Guardiola Múzquiz

GSYC

23 de noviembre de 2022



Parsing ascendente

- Va de las hojas a la raíz (al revés que el descendente)
- Más complicado de entender que el descendente
- Se suelen usar herramientas (generadores de parsers) para escribirlo
- Es más complicado manejar errores
- ¿Por qué usarlo?

Parsing ascendente

- Se puede escribir la gramática limpia (precedencia, recursividad...)
- El generador da ciertas garantías de no ambigüedad (mejor que a mano...)
- Eficiente, se puede hacer en $O(n)$, hay algoritmos que toleran ambigüedad que son $O(n^3)$
- <https://research.swtch.com/yaccalive>

Parsing ascendente: ejemplo

- En el ejemplo vamos a usar un autómata especial con pila
- Si en la cima vemos el lado derecho de una producción la aplicamos al revés (de izquierda a derecha)
- El punto indica los tokens consumidos (ej: $(2. + 7) * 5$), ha consumido dos tokens (2
- Vamos a suponer que hay una tabla de precedencia

Parsing ascendente: ejemplo

El dólar representa el final de la pila *EXPR* es el inicial

%left '+'

%left '*'

EXPR ::= EXPR '*' EXPR |

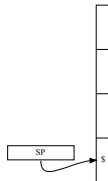
EXPR '+' EXPR |

'(' EXPR ')'

ATOM

ATOM ::= num |

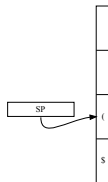
id



Parsing ascendente: ejemplo

Se van haciendo push de tokens.

(3 + 7) * 8

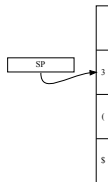


(.3 + 7) * 8

Parsing ascendente: ejemplo

Se van haciendo push de tokens.

(3 + 7) * 8



(**3** . + 7) * 8

Parsing ascendente: ejemplo

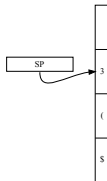
Hasta que en la cima hay algo a la derecha de una regla:

ATOM ::= num

ATOM



(3 + 7) * 8



(**3** . + 7) * 8

Parsing ascendente: ejemplo

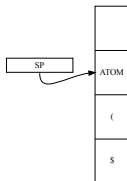
En la cima hay algo a la derecha de una regla:

ATOM ::= num

ATOM



(3 + 7) * 8



(**3** . + 7) * 8

Parsing ascendente: ejemplo

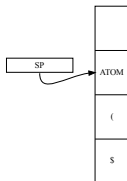
En la cima hay algo a la derecha de una regla:

EXPR ::= ATOM

ATOM



(3 + 7) * 8

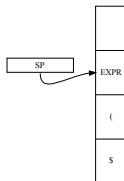
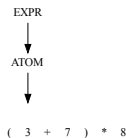


(**3** . + 7) * 8

Parsing ascendente: ejemplo

En la cima hay algo a la derecha de una regla:

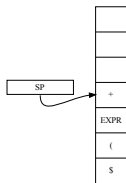
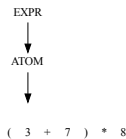
EXPR ::= ATOM



(**3** . + 7) * 8

Parsing ascendente: ejemplo

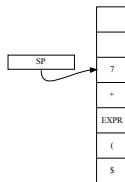
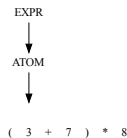
Se sigue haciendo push.



$(3 + .7) * 8$

Parsing ascendente: ejemplo

Se sigue haciendo push.

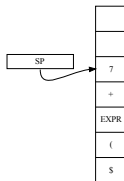
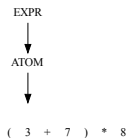


$(3 + \mathbf{7}) * 8$

Parsing ascendente: ejemplo

En la cima hay algo a la derecha de una regla:

ATOM ::= num

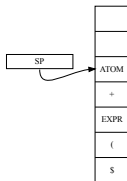
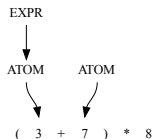


(3+**7**.) * 8

Parsing ascendente: ejemplo

En la cima hay algo a la derecha de una regla:

ATOM ::= num

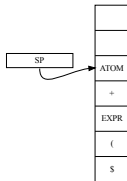
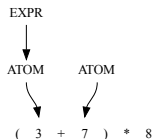


$(3 + \mathbf{7}) * 8$

Parsing ascendente: ejemplo

En la cima hay algo a la derecha de una regla:

EXPR ::= ATOM

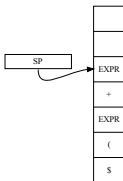
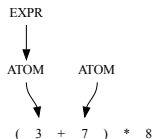


$(3 + \mathbf{7}.) * 8$

Parsing ascendente: ejemplo

En la cima hay algo a la derecha de una regla:

EXPR ::= ATOM

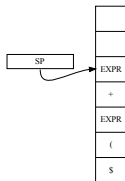
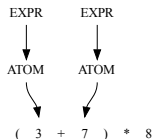


$(3 + 7) * 8$

Parsing ascendente: ejemplo

En la cima hay algo a la derecha de una regla:

$\text{EXPR} ::= \text{EXPR} \text{'+'} \text{EXPR}$

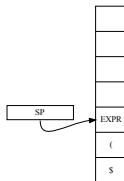
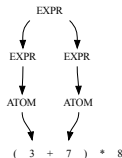


$(3 + \mathbf{7}) * 8$

Parsing ascendente: ejemplo

En la cima hay algo a la derecha de una regla:

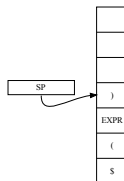
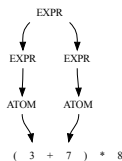
$\text{EXPR} ::= \text{EXPR} \text{'+'} \text{EXPR}$



$(3 + \mathbf{7}) * 8$

Parsing ascendente: ejemplo

Se hace push del paréntesis y de nuevo...

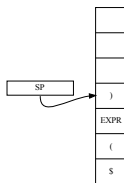
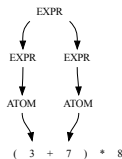


$(3 + 7) * 8$

Parsing ascendente: ejemplo

...en la cima hay algo a la derecha de una regla:

$\text{EXPR} ::= ' (' \text{ EXPR} ') '$

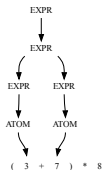


$(3 + 7) * 8$

Parsing ascendente: ejemplo

En la cima hay algo a la derecha de una regla:

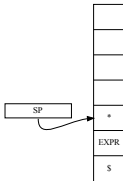
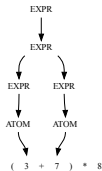
$\text{EXPR} ::= ' (' \text{ EXPR} ') '$



$(3 + 7) \cdot * 8$

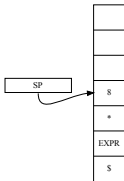
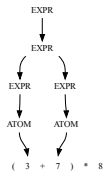
Parsing ascendente: ejemplo

Queda sólo el asterisco...


$$(3 + 7)^* .8$$

Parsing ascendente: ejemplo

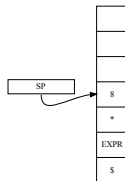
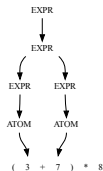
... y el dígito.



$(3 + 7) * \mathbf{8}$.

Parsing ascendente: ejemplo

ATOM ::= num

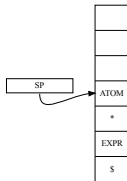
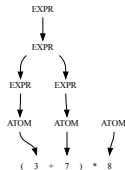


$(3 + 7) * 8.$

Parsing ascendente: ejemplo

En la cima hay algo a la derecha de una regla:

ATOM ::= num

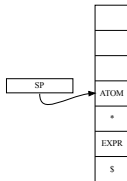
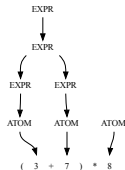


$(3 + 7) * 8.$

Parsing ascendente: ejemplo

En la cima hay algo a la derecha de una regla:

$\text{EXPR} ::= \text{ATOM}$

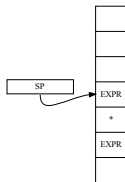
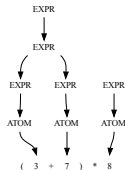


$(3 + 7) * 8.$

Parsing ascendente: ejemplo

En la cima hay algo a la derecha de una regla:

$\text{EXPR} ::= \text{ATOM}$

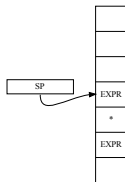
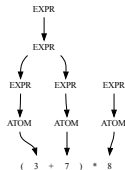


$(3 + 7) * 8.$

Parsing ascendente: ejemplo

En la cima hay algo a la derecha de una regla:

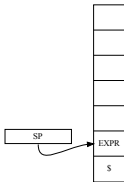
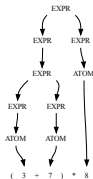
$\text{EXPR} ::= \text{EXPR} \text{'*'} \text{EXPR}$



$(3 + 7) * 8$.

Parsing ascendente: ejemplo

En la pila sólo hay el símbolo inicial $\rightarrow$ aceptamos la entrada



$(3 + 7) * 8.$

Parsing ascendente: ejemplo

- El ejemplo que acabamos de ver es ascendente
- Al ser ascendente por la izquierda es como una derivación LR (**L**eft-to-right, **R**ightmost derivation) marcha atrás
- Con la pila es un ejemplo de shift/reduce (ahora veremos)

Parsing ascendente: ejemplo

- Esto es lo que hemos obtenido
- Es una LR al revés

$(3+7)*8$

$(\text{ATOM}+7)*8$

$(\text{EXPR}+7)*8$

$(\text{EXPR}+\text{ATOM})*8$

$(\text{EXPR}+\text{EXPR})*8$

$\text{EXPR}*8$

$\text{EXPR}*\text{ATOM}$

$\text{EXPR}*\text{EXPR}$

EXPR

Parsing ascendente: ejemplo

- La LR que obtenemos al revés
- Es LR es porque reducción va de derecha a izquierda de producción
- Y ascendente: partimos de los tokens hasta el inicial (descendente: partimos del inicial)
- Funciona con gramáticas recursivas a la derecha, aunque gasta más pila (no como con la implementación descendente, que recurre infinitamente)

EXPR

EXPR*EXPR

EXPR*ATOM

EXPR*8

(EXPR+EXPR)*8

(EXPR+ATOM)*8

(EXPR+7)*8

(ATOM+7)*8

(3+7)*8

Parsing ascendente: nombres

- La regla de producción que se aplica al revés se llama **handle**
- A veces se le llama handle por abreviar, sólo al lado derecho
- Aplicar una producción al revés es una **reducción**
- Se le llama item a una producción a medio procesar, el punto indica por dónde vamos por ejemplo:

EXPR ::= . EXPR '*' EXPR

EXPR ::= EXPR. '*' EXPR

EXPR ::= EXPR '*' . EXPR

EXPR ::= EXPR '*' EXPR.

Parsing ascendente: Shift/Reduce

- La técnica que hemos aplicado en el ejemplo se llama Shift/Reduce, el reconocedor es una autómatas con pila con cuatro operaciones
 - ▶ Shift: leo un token y lo meto en la pila
 - ▶ Reduce: aplico una reducción a un conjunto de elementos en la cima de la pila
 - ▶ Accept: sólo tengo el símbolo inicial en la pila, se acepta la entrada
 - ▶ Error: no puedo aplicar ninguna de las anteriores, se rechaza la entrada

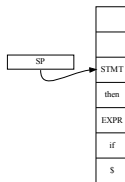
Técnica Shift/Reduce: conflictos

- Si llego a un punto en el que no sé si hacer Shift o Reduce, tengo un conflicto
- Es una forma de ambigüedad de la gramática
 - ▶ Shift/Reduce
 - ▶ Reduce/Reduce

Shift/Reduce

- ¿Reduczo o hago Shift para reducir más adelante?

```
STMT ::= if EXPR then STMT |  
        if EXPR then STMT else STMT |  
...
```



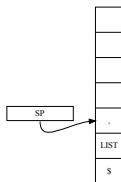
if $x == 3$ then **f**(3) .else

Reduce/Reduce

- ¿Aplico la tercera de *LIST* o la primera de *OPTCOMMA*?

```
LIST ::= id|  
      LIST OPTCOMMA id|  
      LIST ','
```

```
OPTCOMMA ::= ', '|  
          EMPTY
```



a , **b** .

Conflictos

- Dado un conjunto de items, desde el punto
- Shift/Reduce: Una regla es un prefijo de otra (a la izquierda del punto)
- Reduce/Reduce: Una regla es un sufijo de otra (a la derecha del punto)

Tipos de parser y gramáticas

- Hay varios tipos de parsers bottom-up
- Los parsers que veremos, hacen **shift de estados**, (no como el Shift/Reduce, que hace de símbolos)
- Vamos a construir un autómatas que se llamara SLR (Simple LR)
- SLR es un tipo de gramáticas que **no es suficientemente expresivo**
- Luego construiremos otro LR(k) (Left-to-Right, Rightmost, k tokens lookahead)
- Finalmente construimos LALR(k) (LA es de lookahead ¡De nuevo!) que genera **muchos menos estados**
- Cada uno de estos tiene su familia de gramáticas asociadas con diferente poder expresivo
- Yacc construye parsers LALR (cuando no se pone nada es como si hubiese un 1, es decir LALR(1))

Tipos de parser y gramáticas

- Todas un subconjunto estricto de CFG, libres de contexto, las de los lenguajes de programación estándar
- Hay gramáticas ambiguas que son CFG que están fuera de todas estas familias *LR* de gramáticas
- Conflictos Shift/Reduce Reduce/Reduce $\rightarrow$ ambigua $\notin$ LR(k)
- $LL(0) \subset LL(1) \subset LL(k)$
- $LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1) \subset LR(k)$
- $LL(k) \subset LR(k)$
- Hay construcciones habituales que no son SLR (ver ejemplo 4.48 del dragón)

First y follow

- Útil en general para razonar sobre parsers (también top-down)

First

- $z \in \text{First}(\alpha)$ para α cualquier cadena de símbolos de la gramática
- Conjunto de terminales que son el primer símbolo a la derecha de una derivación $\alpha \xrightarrow{*} zB$
- B puede ser cualquier cosas incluido ϵ

First

```
EXPR ::= EXPR '*' EXPR |  
        EXPR '+' EXPR |  
        '(' EXPR ')' |  
        ATOM  
ATOM ::= num |  
        id
```

- $First(EXPR) = \{ '(', num, id \}$

Follow

- $z \in \text{Follow}(A)$ donde A es un no terminal
- Si z aparece justo después de A en una derivación
- $X \xrightarrow{*} WAzY$

Follow

$$\begin{aligned} \text{EXPR} ::= & \text{EXPR } '*' \text{ EXPR} \mid \\ & \text{EXPR } '+' \text{ EXPR} \mid \\ & '(' \text{ EXPR } ') ' \mid \end{aligned}$$
$$\begin{aligned} \text{ATOM} ::= & \text{num} \mid \\ & \text{id} \end{aligned}$$

- $\text{Follow}(\text{EXPR}) = \{')', +, *, \$\}$
- El dólar significa el final de la entrada de la regla

Cierre

- Dado un conjunto de items:

`EXPR ::= EXPR '*' .EXPR`

`EXPR ::= .ATOM`

- El cierre (Closure) es poner coger los no terminales que siguen al punto
- Y añadir la regla con el punto a la izquierda

`EXPR ::= EXPR '*' .EXPR`

`EXPR ::= .ATOM`

`ATOM ::= .num`

`ATOM ::= .id`

SLR

```
EXPR ::= EXPR '*' EXPR |  
        EXPR '+' EXPR |  
        '(' EXPR ')' |  
        ATOM  
ATOM ::= num |  
        id
```

- Vamos a construir un autómata SLR para esta gramática
- A diferencia del shift/reduce, mete estados en la pila, el actual es el que está en la cima
- Va a tener estados y acciones según el siguiente token de lookahead
- Las acciones son reduce, shift, accept, error
- Las tablas tienen que tener entradas del tipo $action(s_1, num) = shift$ con s_1 un estado y num un token de entrada

SLR

```
EXPR' ::= EXPR  
EXPR ::= EXPR '*' EXPR |  
        EXPR '+' EXPR |  
        '(' EXPR ')' |  
        ATOM  
ATOM ::= num |  
        id
```

- Primero se añade una regla extra a la gramática para el símbolo inicial

SLR: conjunto de items

- Se construyen el conjunto canónico de items $LR(0)$
- Se comienza con $Closure(EXPR' ::= .EXPR)$
- Para cada símbolo de la gramática se calculan los siguientes estados
- Se calcula la cerradura de cada uno de estos estados

SLR: items

- Se construyen los Items $LR(0)$
- Se comienza con $I_0 = \text{Closure}(\text{EXPR}' ::= \text{.EXPR})$

$\text{EXPR}' ::= \text{.EXPR}$

$\text{EXPR} ::= \text{.EXPR '}' \text{EXPR} \mid$
 $\text{.EXPR '+' EXPR} \mid$
 $\text{.'(' EXPR ')'} \mid$
 .ATOM

$\text{ATOM} ::= \text{.num} \mid$
 .id

SLR: items

- I_1 , llega $EXPR$, calculo los items...

$EXPR' ::= EXPR.$

$EXPR ::= EXPR. \text{'*'} EXPR \mid$
 $EXPR. \text{'+' } EXPR$

SLR: items

- I_1 , llega $EXPR$, calculo los items...
- ...y su cierre

$EXPR' ::= EXPR.$

$EXPR ::= EXPR. \text{'*'} EXPR \mid$
 $EXPR. \text{'+' } EXPR$

SLR: items

- I_2 , llega '(' a I_1 , calculo los items...
- ...y su cierre

EXPR ::= '(' .EXPR ')'

EXPR ::= .EXPR '*' EXPR |
 (EXPR '+' EXPR |
 '(' EXPR ')') |
 .ATOM

ATOM ::= .num |
 .id

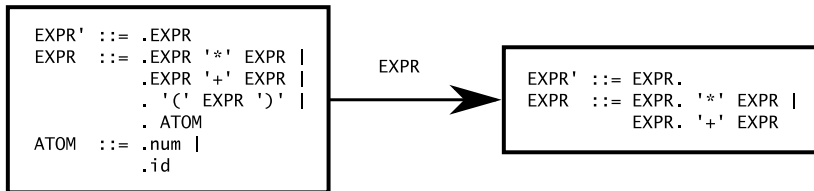
SLR: items

Se van añadiendo las transiciones por shift de símbolos

```
EXPR' ::= .EXPR  
EXPR  ::= .EXPR '*' EXPR |  
        .EXPR '+' EXPR |  
        . '(' EXPR ')' |  
        . ATOM  
ATOM  ::= .num |  
        .id
```

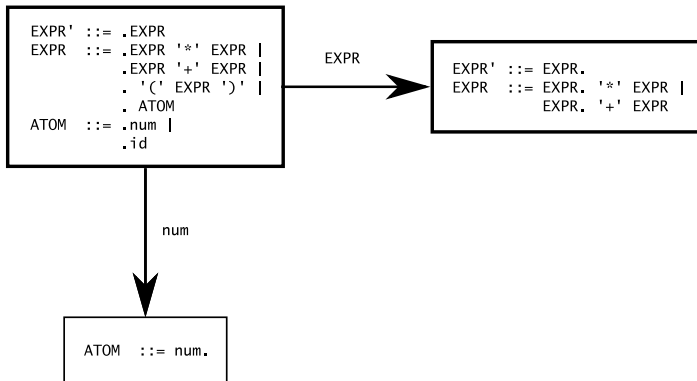
SLR: items

Se van añadiendo las transiciones por shift de símbolos



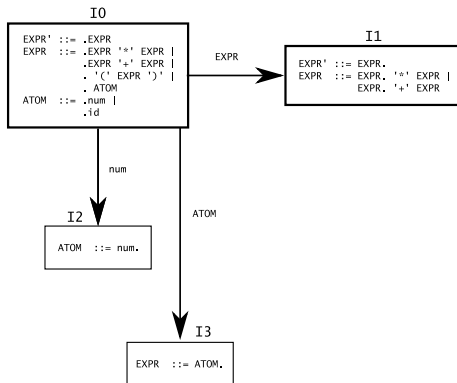
SLR: items

Se van añadiendo las transiciones por shift de símbolos



SLR: items

Se van añadiendo las transiciones por shift de símbolos



- Al final, tengo todos los estados (se deja como ejercicio)
- Y sus transiciones

- Las transiciones de ese grafo serán los shifts
- Los reduce volverán atrás el punto después de hacer la reducción (luego más)

SLR

Para la gramática

$$E ::= E \text{ '+' } T \mid$$
$$T$$
$$T ::= T \text{ '*' } F \mid$$
$$F$$
$$F ::= \text{'(' } E \text{ ')'} \mid$$
$$\text{id}$$

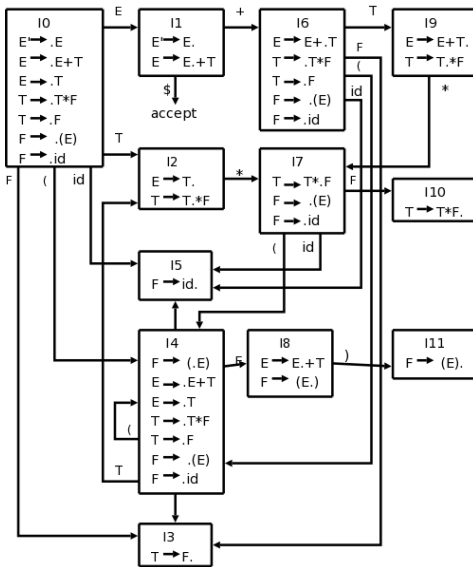
SLR

Se extiende

$$E' ::= E$$
$$E ::= E \text{'+' } T \mid$$
$$T$$
$$T ::= T \text{'*'} F \mid$$
$$F$$
$$F ::= \text{'(' } E \text{'')} \mid$$
$$\text{id}$$

SLR: ejemplo completo

Se construye el grafo



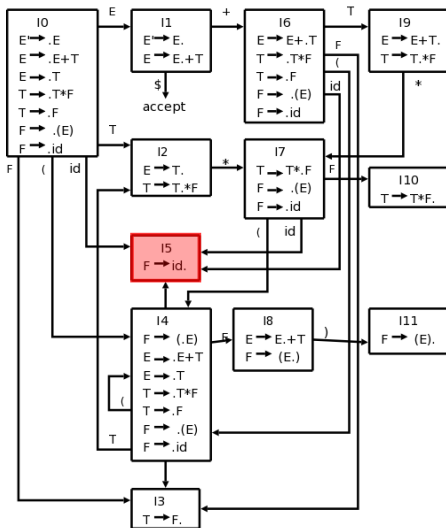
Máquina de estados: reduce

- En una reducción el punto vuelve atrás
- $X \rightarrow id.$ terminaría en $Z \rightarrow .X$
- En la máquina de estados es ir hacia atrás en los símbolos de la derecha de la regla
- Y avanzar en los símbolos que hay en la pila

Máquina de estados: reduce

Ejemplo

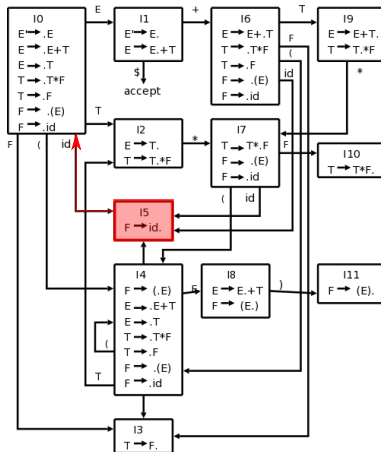
- ① Estoy en el estado *I5*, con $T * id$ en la pila de símbolos



Máquina de estados: reduce

Ejemplo

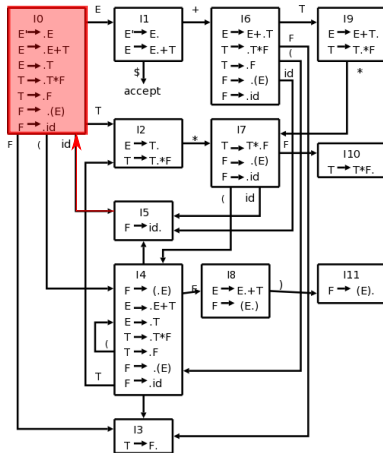
- 1 Estoy en el estado *I5*, con $T * id$ en la pila de símbolos
- 2 Reduzco con $F \rightarrow id$, sigo hacia atrás la flecha de *id* en la máquina de estados hasta *I0*



Máquina de estados: reduce

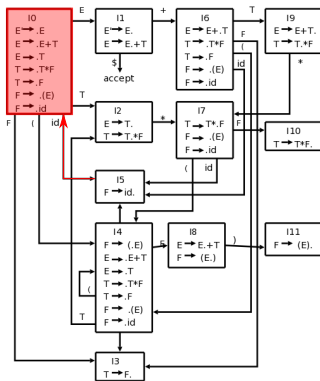
Ejemplo

- 1 Estoy en el estado *I5*, con $T * id$ en la pila de símbolos
- 2 Reduzco con $F \rightarrow id$, sigo hacia atrás la flecha de id en la máquina de estados hasta *I0*



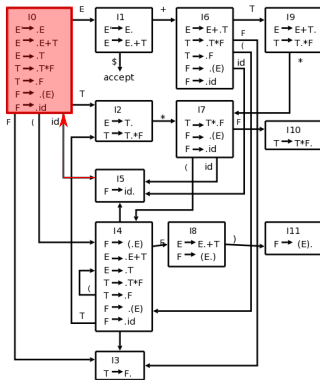
Máquina de estados: reduce

- 1 Estoy en el estado $I5$, con $T * id$ en la pila de símbolos
- 2 Reduzco con $F \rightarrow id$, sigo hacia atrás la flecha de id en la máquina de estados $I0$
- 3 Ahora tengo $T * F$ en la pila, sigo eso en la máquina de estados $I0 \rightarrow I2 \rightarrow I7 \rightarrow I10$
- 4 Acabo en ese estado, codifico en una tabla



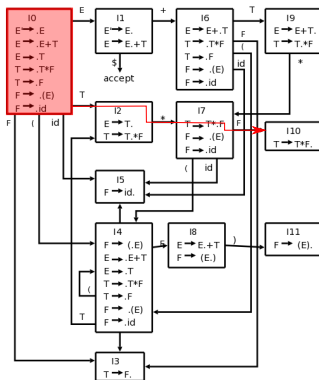
Máquina de estados: reduce

- 1 Estoy en el estado $I5$, con $T * id$ en la pila de símbolos
- 2 Reduzco con $F \rightarrow id$, sigo hacia atrás la flecha de id en la máquina de estados $I0$
- 3 Ahora tengo $T * F$ en la pila, sigo eso en la máquina de estados $I0 \rightarrow I2 \rightarrow I7 \rightarrow I10$
- 4 Acabo en ese estado, codifico en una tabla



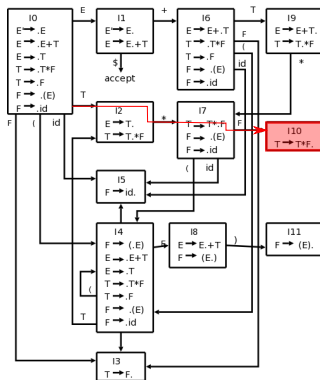
Máquina de estados: reduce

- 1 Estoy en el estado $I5$, con $T * id$ en la pila de símbolos
- 2 Reduzco con $F \rightarrow id$, sigo hacia atrás la flecha de id en la máquina de estados $I0$
- 3 Ahora tengo $T * F$ en la pila, sigo eso en la máquina de estados $I0 \rightarrow I2 \rightarrow I7 \rightarrow I10$
- 4 Acabo en ese estado, codifico en una tabla



Máquina de estados: reduce

- 1 Estoy en el estado $I5$, con $T * id$ en la pila de símbolos
- 2 Reduzco con $F \rightarrow id$, sigo hacia atrás la flecha de id en la máquina de estados $I0$
- 3 Ahora tengo $T * F$ en la pila, sigo eso en la máquina de estados $I0 \rightarrow I2 \rightarrow I7 \rightarrow I10$
- 4 Acabo en ese estado, codifico en una tabla



SLR: tabla

- Añado las acciones
- Si hay una transición a I_k con un terminal a ,
 $Action(S_j, a) = SHIFTS_k$, avanza a push del estado k
- $Action(S_j, z) = REDUCE A \rightarrow B$ si $A \rightarrow B.$ es un un item de I_j , salvo $EXPR'$
- Si reduzco la regla de $EXPR'$ que es especial, entonces aplico
 $Action(S_j, z) = ACCEPT$
- Si si hay una transición a I_k con un no terminal,
 $Action(i, S_j) = GOTO S_k$, es decir push del estado k
- Si me quedo sin nada que hacer *Error*
- Si no puedo decidir entre alguna de las anteriores al crear la tabla, conflicto

SLR: ejemplo completo

Se construye la tabla de transiciones

tabla

Estado	ACTION						GOTO		
	id	+	*	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				accept			
2		r ²	s7		r ²	r ²			
3		r ⁴	r ⁴		r ⁴	r ⁴			
4	s5			s4			8	2	3
5		r ⁶	r ⁶		r ⁶	r ⁶			
6	s5		s4					9	3
7	s5		s4						10
8		s6			s11				
9		r ¹	r ⁷		r ¹	r ¹			
10		r ³	r ³		r ³	r ³			
11		r ⁵	r ⁵		r ⁵	r ⁵			

1 $E \rightarrow E + T$

2 $E \rightarrow T$

3 $T \rightarrow T * F$

4 $T \rightarrow F$

5 $F \rightarrow (E)$

6 $F \rightarrow id$

OJO:

- Reduce hace pop de n símbolos de la pila de estados (Ejemplo $E \rightarrow E + T$, hago pop de 3 estados y aplico el GOTO de E)
- Si el elemento actual es terminal, miro el estado, aplico acción
- Si es no terminal, miro el estado aplico GOTO

SLR: ejemplo completo

Ejecución para $id * id + id\$$

Pila de estados	Símbolos	Entrada	Action
0		$id * id + id\$$	shift
0 5	id	$*id + id\$$	reduce $F \rightarrow id$
0 3	F	$*id + id\$$	reduce $T \rightarrow id$
0 2	T	$*id + id\$$	shift
0 2 7	T^*	$id + id\$$	shift
0 2 7 5	T^*id	$+id\$$	reduce $F \rightarrow id$
0 2 7 10	T^*F	$+id\$$	reduce $T \rightarrow T * F$
0 2	T	$+id\$$	reduce $E \rightarrow T$
0 1	E	$+id\$$	shift
0 1 6	$E+$	$id\$$	shift
0 1 6 5	$E+id$	$\$$	reduce $F \rightarrow id$
0 1 6 3	$E+F$	$\$$	reduce $T \rightarrow F$
0 1 6 9	$E+T$	$\$$	reduce $E \rightarrow E + T$
0 1	E	$\$$	accept

Pila de estados

- El estado en la cima de la pila es el estado actual

GOTO

- Hago push del estado a la pila
- Se aplica como parte de un reduce

Shift

- Tomo un token de la entrada, lo meto en la pila, hago push de un estado (cambio el estado actual)

Shift: ejemplo

- Por ejemplo, estoy en el estado 7, me llega *id*, la tabla dice shift de 5
- Saco el token de la entrada
- Hago push de *id* a la pila de símbolos y push de 5 a la pila de estados

Pila de estados	Símbolos	Entrada	Action
0 2 7	T*	<i>id</i> + <i>id</i> \$	shift
0 2 7	T*	+ <i>id</i> \$	paso intermedio: lex <i>id</i>
0 2 7 5	T* <i>id</i>	+ <i>id</i> \$	paso intermedio: push <i>id</i> y push 5
0 2 7 5	T* <i>id</i>	+ <i>id</i> \$	

Reduce

- La reducción se aplica sobre una regla (el número es la regla que se aplica)
- Igual que en el ejemplo, reduce hace pop de n símbolos de la pila de símbolos
- Después hace push del símbolo que hay a la izquierda de la regla
- Aplico el GOTO asociado al símbolo que he dejado en la pila (hago push del estado)

Reduce: ejemplo

- Por ejemplo, estoy en el estado 2, y hago reduce de la regla 2 (tengo $+$ a la entrada)
- Hago pop de T y de 2 (el lado izquierdo de la regla de derivación)
- Hago push de E a la pila de símbolos
- En el estado 0 y con E , hago GOTO a 1

Pila de estados	Símbolos	Entrada	Action
0 2	T	$+id\$$	reduce $E \rightarrow T$
0		$+id\$$	paso intermedio: pop 2 y pop T
0	E	$+id\$$	paso intermedio: push E
0 1	E	$+id\$$	paso intermedio: GOTO 1
0 1	E	$+id\$$	

Reduce: ejemplo 2

- Por ejemplo, estoy en el estado 10, y hago reduce de la regla 3 (tengo + a la entrada)
- Hago pop de $T * F$ y de 7 10
- Paso a estar en el estado 0 que se ha quedado descubierto, ejecuto el GOTO 2

Pila de estados	Símbolos	Entrada	Action
0 2 7 10	$T * F$	$+id\$$	reduce $T \rightarrow T * F$
0		$+id\$$	paso intermedio: pop 2, 7 10, pop $T * F$
0		$+id\$$	paso intermedio: push T , GOTO 2
0 2	T	$+id\$$	

LR, LALR

- $LR(k)$: items anotados por lookahead $LR(k)$ (no sólo a partir de $LR(0)$) como SLR
- $LALR$: reducción de estados, estados con items $LR(k)$ reducidos casi a $LR(0)$
- Son fundamentalmente parecidos los autómatas
- Yacc usa $LALR$, con lo que hemos visto aquí se puede entender
- *First* y *Follow* fundamental para entender los problemas de ambigüedad en bottom-up y en top-down

- SLR usaba el conjunto de conjuntos de items $LR(0)$ que hemos construido
- $LR(1)$ (LR canónico) usa el conjunto canónico $LR(1)$ que es el $LR(0)$ pero anotado por los terminales de lookahead
- Esto es importante si el punto está a la derecha del todo, que significa que es una reducción si el leído está en *FOLLOW*
- En $LR(1)$ se hace la reducción si el token leído es justo la anotación
- Esto hace que se permitan más lenguajes

- Juntamos varios de los estados de un $LR(1)$ consiguiendo casi SLR , pero con los extras de expresividad
- Sencillamente tiene muchos menos estados

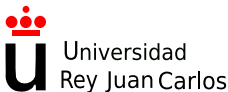
Tema 7: Yacc

Autómatas y desarrollo avanzado de software

Gorka Guardiola Múzquiz

GSYC

23 de noviembre de 2022



- Yacc, “Yet Another Compiler Compiler”, generador de parsers basado en LALR
- A principio de los 70 Stephen C. Johnson en Bell Labs (AT&T), basándose en trabajo de Knuth
- Formó la base del compilador de Unix y C
- En Linux, clon de Unix, hay un clon de yacc: bison

Historia

- Yacc original escrito en C para C
- Se reescribió para Plan 9, luego para Inferno en Limbo
- Goyacc transcripción del de Inferno para Go
- Se usó en las primeras versiones del compilador (ahora descendente recursivo)

Goyacc

- Prácticamente idéntico en espíritu que yacc
- Se ha sacado de la distro oficial, es una herramienta separada
- <https://godoc.org/golang.org/x/tools/cmd/goyacc>
- `go get -u golang.org/x/tools/cmd/goyacc`

- El lexer cumple un interfaz concreto
- Se escriben acciones para las reglas de producción en un fichero `xxxx.y`
- Se ejecuta `goyacc` sobre `xxxx.y` y genera entre otras cosas un fichero en Go
- Los valores para los tokens los genera el propio yacc

Fichero goyacc: formato

```
1 %{
2     //Go
3 }%
4 %union {
5     //Campos para el simbolo
6     sval    string
7     ival    int
8     stmt    *Statement
9 }
10 //precedencia de operadores
11 %left    ',', ' '
12 %left    OR AND
13
14 //tipos de token
15 %token   ARRAY CASE CONSTS DEFAULT DO ELSE
16 //Campos del simbolo
17 %token   <ival> INT CHAR
18 %token   <sval> STR
19
20 //tipos y campo de no terminales
21 %type    <stmt> stmt body assignstmt
```


Fichero goyacc: formato

Reglas de la gramática:

```
22 %%
23 prog:
24     proghdr decls
25     {
26         if env.prev != nil { panic("missing popenv()") }
27         if debug['P'] != 0 { dumpprog(os.Stderr, env.prog) }
28     }
29     ;
30 decls:
31     decls decl
32     |
33     decl
34     ;
```

Fichero goyacc: formato

```
35 proghdr:
36 |
37     ID ID ';'
38     {
39         env.prog = newprog($2)
40     }
41 |
42     PROGRAM ID ';'
43     {
44         env.prog = newprog($2)
45     }
46 ;
47 type:
48     TYPEID
49     {
50         $$ = $1.ttype
51     }
52 |
53     newtype
54 ;
55 list : /* empty */
56 | list stmt '\n'
57 ;
58 %%
59 //Go
```

- El lexer cumple un interfaz concreto
- Se escriben acciones para las reglas de producción en un fichero `xxxx.y`
- Se ejecuta `goyacc` sobre `xxxx.y` y genera entre otras cosas un fichero en Go
- Los valores para los tokens los genera el propio yacc

- Se puede escribir código en varios sitios por razones históricas
- En C el orden de las funciones y declaraciones globales importa (en Go, no)
- Las reglas de producción acaban en punto y coma
- Acciones entre llaves

- El lexer tiene que tener un interfaz, yySymType tiene los campos definidos en %union.

```
1 type yyLexer interface {  
2     Lex(lval *yySymType) int  
3     Error(e string)  
4 }
```

- Igual que el lexer, todas las cosas definidas por yacc empiezan por yy,
- Si quiero meter en otro paquete (útil) puedo pasarle el parámetro `-p XXX`
- Del *lookahead* se ocupa la implementación que genera yacc
- Por ejemplo el parser tendrá que ser en ese caso:

```
1 type XXXParser interface {  
2     Parse(XXXLex) int  
3     Lookahead() int  
4 }
```

- Yacc imprime tabla de parsing en y.output (o zzz.out) con el parámetro `-v zzz.out`

```
1 state 0
2   $accept: . list $end
3   list: . (1)
4
5   . reduce 1 (src line 44)
6
7   list goto 1
8
9 state 1
10  $accept: list.$end
11  list: list.stmt \n
12
13  $end accept
14  DIGIT shift 8
15  LETTER shift 4
16  — shift 6
17  ( shift 5
18  . error
19
20  expr goto 3
21  number goto 7
22  stmt goto 2
23  ...
```

- Todo junto
- `goyacc -p Blaster -v table.out blast.y`
- El fichero de estados estarán en `table.out`
- El fichero en go será `y.go`
- El interfaz del lexer será `BlasterLex`
- La variable del lexer, será `blasterlex`

- Vamos a ver un ejemplo en varios paquetes
- El lexer y la tabla de símbolos en otro paquete (`lex` y `sym`)
- Vamos a tener que adaptarlo, sobre todo por los valores de los tipos de token
- Estaba pensado para ir todo en un paquete, separarlo es un poco raro
- El estado de parsing va en la variable del lexer

- Es una calculadora con todos los números en float (como en Javascript, no lo hagáis, mala idea)
- Vamos a programar un intérprete con atributos sintéticos (no construye AST)
- Tiene variables
- Una línea es una expresión o una asignación
- Cuando se evalúa una expresión se imprime su valor
- Una entrada puede ser `x=3.0\n(x *4.0)+5.0\n`

- Para compilar y ejecutar
- `goyacc -p calc -v st.ot calc.y && go install && asc`
- En el fichero de yacc se escribe un comentario para que lo ignore el comando go
- Fuera de la generación de código (para que yacc no lo meta en el fichero de go)
- `// +build ignore`

Goyacc

- La gramática se escribe en BNF pero con notación un poco diferente
- Acaba en ;

`<LIST> := empty | <LIST> <STMT> '\n'`

Se escribe:

```
list :  
    | list stmt '\n'  
    ;
```

- Las acciones van entre llaves

- Evitar la recursividad por la derecha
- Para usar menos pila

Calculadora

```
1 // +build ignore
2 %{
3
4 package main
5
6 import (
7     "fmt"
8     "comp/asc/lex"
9     "comp/asc/sym"
10    "log"
11 )
12 %}
```

Calculadora

```
13
14 %union{
15     val float64      //float
16     name string //var
17 }
18
19 %token <val> NUMBER /*campo val : float64 */
20 %type <val> expr      /*campo val : float64 */
21 %token <name> VAR      /*campo name : string */
22
23 %left '|'
24 %left '&'
25 %left '+' '-'
26 %left '*' '/' '%'
27 %unary UMINUS
28
29 %start list /* inicial , el primero si no estuviese */
30
31 %%
```


Calculadora

- La unión serán campos del record de terminales y no terminales
- Se expanden como \$\$, \$1, \$2...
- El tipo lo da %token y %type, dependiendo de si son terminales o no terminales
- Las definiciones de token definen constantes que serán el tipo de token (NUMBER, VAR)
- El resto de tokens serán enteros (runas), como '*'
- La precedencia la dan los left/right/unary y el orden de definición, más abajo, más precedencia
- El signo menos (-) aparece varias veces y se usa el truco de precedencia contextual (en diferentes reglas diferente precedencia)
- Eso es lo que significa el %prec MINUS en la regla de la gramática (más adelante)

Calculadora

```
32
33
34 list      : /* empty */
35           | list stmt '\n'
36           ;
37
38 stmt      :      expr
39           {
40             fmt.Printf( "%v\n", $1 )
41           }
42           | VAR '=' expr
43           {
44             cl, _ := calclex.(*calcLex)
45             s, err := cl.parseSt.stkEnv.NewSym($1, sym.SVar)
46             if err != nil {
47               panic(err)
48             }
49             s.FloatVal = $3
50           }
```

Calculadora

```
51 expr      :      '(' expr ')'  
52          { $$ = $2 }  
53 |  
54          expr '+' expr  
55          { $$ = $1 + $3 }  
56 |  
57          expr '-' expr  
58          { $$ = $1 - $3 }  
59 |  
60          expr '*' expr  
61          { $$ = $1 * $3 }  
62 |  
63          expr '/' expr  
64          { $$ = $1 / $3 }  
65 |  
66          expr '%' expr  
67          { $$ = float64(int($1) % int($3)) }  
68 |  
69          expr '&' expr  
70          { $$ = float64(int($1) & int($3)) }  
71 |  
72          expr '|' expr  
73          { $$ = float64(int($1) | int($3)) }  
74 |  
75          '-' expr      %prec UMINUS  
76          { $$ = -$2 }
```

Calculadora

```
69 |     VAR
70 |     {
71 |         cl, _ := calclex.(* calcLex)
72 |         s := cl.parseSt.stkEnv.GetSym($1)
73 |         if s == nil {
74 |             panic("undefined")
75 |         }
76 |         $$ = s.FloatVal
77 |     }
78 |     NUMBER
79 |     { $$ = $1 }
80 | ;
81 %%
```

Ojo, si quiero poder poner $p = p + 1$

```
69 |     VAR
70 |     {
71 |         dprintf("stat: VAR = EXPR\n")
72 |         cl, _ := calclex.(*calcLex)
73 |         s := cl.parseSt.stkEnv.GetSym($1);
74 |         var err error
75 |         if s != nil && s.SymType() != sym.SVar {
76 |             fmt.Fprintf(os.Stderr, "Assign to %s, not lval\n", $1)
77 |             cl.nErr++
78 |             s = nil
79 |         }
80 |         if s == nil {
81 |             s, err = cl.parseSt.stkEnv.NewSym($1, sym.SVar)
82 |             if err != nil {
83 |                 panic(err)
84 |             }
85 |         }
86 |     }
87 |     NUMBER
88 |     { $$ = $1 }
89 | ;
%%
```

Calculadora

- Las acciones aparecen entre llaves
- Ojo, \$1 se expande a `calcDollar[1].val`
- Es como una macro, no sabe nada, poner paréntesis

Calculadora: lexer

- Pensado para ir en el mismo paquete
- Por los enteros para los tipos de los tokens, i.e. los terminales (definidos con `%token`)
- Se puede adaptar

Calculadora

```
82 type parserState struct {  
83     depth int  
84     stkEnv sym.StkEnv  
85 }  
86  
87 type calcLex struct {  
88     origLexer *lex.Lexer  
89     parseSt *parserState  
90 }  
91 func (l *calcLex) Lex(lval *calcSymType) int {  
92     tok, err := l.origLexer.Scan()  
93     if err != nil {  
94         panic(err)  
95     }  
96     switch tok.Type {  
97     case lex.TokFloatVal:  
98         lval.val = tok.TokFloatVal  
99         return NUMBER
```


Calculadora

```
100     case lex.TokId:
101         lval.name = tok.Lexema
102         return VAR
103     case lex.TokEof:
104         return -1
105     case lex.TokBad:
106         panic("error, bad token")
107     default:
108         return int(tok.Type)    //trick
109     }
110 }
```

Calculadora

```
111 func (l *calcLex) Error(e string) {  
112     panic(e)  
113 }  
114  
115 func NewParserState() *parserState {  
116     p := &parserState{}  
117     p.stkEnv.PushEnv() //global  
118     //p.initSyms()  
119     p.stkEnv.PushEnv()  
120     return p  
121 }  
122 func SetDebug(debLev int) {  
123     calcDebug = debLev  
124 }
```

Calculadora

- Mejor si se pone main en otro paquete
- Mejor para testing y demás

```
1 func main() {
2     SetDebug(4)
3     s := "x=3.0\n(x *4.0)+5.0\n"
4
5     var nl = &calcLex{}
6     l, err := lex.NewStringLexer(s)
7     if err != nil {
8         log.Fatal(err)
9     }
10    nl.Lexer.origLexer = l
11    ps := NewParserState()
12    nl.parseSt = ps
13
14    p := calcNewParser()
15    p.Parse(nl)
16 }
```

Calculadora

- La variable `xxxDebug` para depuración del parser
- 1 Error, 2 Error recuperación, 3 lex info, 4 push y pop de estados en la pila

Calculadora

- **char tok** es que se ha leído un token (shift) y el estado en que está
- **reduce** significa reduce qué regla reduce
- Los estados se pueden consultar en la tabla

```
char tok--1 in state-0
```

```
reduce 1 in:
```

```
state-0
```

```
char tok--1 in state-1
```

```
lex VAR(57347)
```

```
char tok--1 in state-4
```

```
lex '='(61)
```

```
char tok--1 in state-16
```

```
lex NUMBER(57346)
```

Calculadora

- Si se ejecuta `goyacc -p nombre -v st.ot parser.y`
- El fichero de estados **st.ot**
- Tiene los items, las acciones y los gotos en ese orden
- En las acciones el punto es cualquier cosa
- Cuando es reduce viene la regla (`src line XXX`)
- **shift** hace shift de un token y salta a un estado
- **reduce** (con la regla anotada en el código fuente o en el item) hace pop de n estados y luego se aplica el goto en el nuevo estado que se descubre

Calculadora

```
state 0
    $accept: .list $end
    list: .      (1)

    . reduce 1 (src line 37)

    list goto 1
state 1
    $accept: list.$end
    list: list.stat '\n'

    $end accept
    NUMBER shift 7
    VAR shift 4
    '-' shift 6
    '(' shift 5
    . error

    expr goto 3
    stat goto 2

...
```

- Inclusión de acciones modifica la gramática

A: B C {...}

;

No es

A: B {...} C

;

- Introduce un símbolo, (*c* es \$3)

```
a : b {..} c  
;
```

Equivale a añadir una nueva regla

```
a : b temp c  
;
```

```
temp : /*empty*/  
{ ...}  
;
```

- Las acciones se van a aplicar siguiendo la derivación: de forma ascendente
- Hay que poner los push en una acción previa de la misma regla para que se haga antes

```
1 functype:
2     FUNCTION
3     {
4         PushEnvironment()
5     }
6     '(' parms ')' ':' funcret
7     {
8         $$ = newtype(Tfunc)
9         $$ . parms = $4
10        $$ . rtype = $7
11        PopEnvironment()
12    }
```

Calculadora: manejo de errores

- El parser llama al método `Error` del lexer cuando hay un error
- Y luego intenta recuperarse
- Típicamente se tolera un número de errores y luego se muere el programa

Calculadora: manejo de errores

- Cuando yacc encuentra un error llama al método Error del lexer
- La funcion puede indicar que se ha zanjado el Error poniendo `Errflag = 0`
- Aparte hay reglas de producción de error, que encajan con un error y cualquier cosa

Calculadora

En el ejemplo de arriba, ponemos de entrada:

`x=3.0\n(x *)+5.0\n 3.0+ 2.0\n` Se imprime el error y acaba:

```
1 /fakefile:2 near []
```

Queremos que procese mas líneas, añadimos una entrada error y nos recuperamos

```
1 list      : /* empty */  
2 | list stat '\n'  
3 | error '\n'  
4 | {  
5 |     Errflag = 0;  
6 |     fmt.Println("recover") //so I can see it  
7 | }  
8 ;
```

Sigue ejecutando las siguientes líneas

```
1 /fakefile:2 near []  
2 recover  
3 5
```

Calculadora

- Cuando arreglar el problema y resincronizarse es complicado
- Sólo el primer error suele ser completamente fiable
- Se hace lo que se puede
- A veces son inevitables los errores en cascada
- Si estoy contruyendo el AST, es normal rellenar con un nodo erróneo

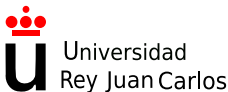
Tema 8: Tipos de datos

Autómatas y desarrollo avanzado de software

Gorka Guardiola Múzquiz

GSYC

23 de noviembre de 2022



Comprobaciones semánticas

- Ya vimos declaraciones de variables
- A los símbolos se les puede añadir más información (declarando tipos o con tipos implícitos)
- Y comprobar: en compilación, **estático** o en ejecución, **dinámico**
- Hay varias partes que hay que tratar:
 - ▶ Tipos **primitivos**
 - ▶ Tipos **definidos por el usuario**
 - ▶ **Literales**, tipos primitivos/definidos por el usuario
 - ▶ Tipos **universales**
 - ▶ **Comprobación** de tipos
 - ▶ **Inferencia de tipos** (expresiones, literales)

Tipos

- De momento vamos a hablar de **tipado estático**
- Los tipos son atributos sintéticos
- Son comprobaciones semánticas, se hacen:
 - ▶ Al construir el AST
 - ▶ En una pasada posterior sobre el AST

Tipos primitivos

- Aunque no hubiese tipos definidos por el usuario
- Expresiones, asignaciones, comprobar los tipos
- Tengo literales (que forman parte de expresiones)

Tipos primitivos

- Cuando evalúo una expresión, hago una asignación o paso parámetros a una llamada
- Ojo, el valor es normalmente de ejecución, el tipo es de compilación (...o no, tipado estático vs. dinámico)
- Sea con el AST o directamente
- Igual que un valor constante es un atributo sintético...
- ...los tipos también lo son, con el extra de que...
- Para los tipos hay que comprobar si son compatibles (y si hay cast o inferencia de tipos)

Tipos primitivos

- Para las variables/parámetros, el símbolo tiene el tipo
- Puedo hacer algo parecido para las constantes con el símbolo que pongo en el AST
- La compatibilidad la da el lenguaje (¿puedo sumar un int a un float? ¿hay castings? ¿son posibles?)
- Ojo, en un operador binario, por ejemplo hay tres tipos en juego, el operador y los dos operandos
- Un casting es una función/operador predefinido que recibe un tipo y devuelve otro

Tipos primitivos

- Básicamente tengo **una tabla** con los tipos que soporto
- Tengo una tabla que me permite definir tipos simples
- Ahí irán luego también los tipos definidos por el usuario
- Los tipos tienen un nombre, y la misma tabla se usa para mejorar los tokens (de `id` a `typeId`)
- Es como una tabla de símbolos, pero de tipos (se pueden luego definir nuevos)
- Hay simples (`int`, `float`, `char`...) y compuestos o agregados (`array`, `string`, registros, punteros a función o closures...)

Ejemplo

- Calculadora con tipos como atributos sintéticos
- Los tipos de datos se tratan de forma similar a los valores al evaluar una expresión
- Cada valor en la calculadora (i.e. 3.8) es un símbolo con valor y tipo (i.e. 3.8, **float**)
- En cada operación se comprueba la compatibilidad de tipo

Ejemplo

- Vamos a ver un ejemplo mínimo
- Los tipos no tienen estructura, sólo hay float e int

Ejemplo

- Lo primero es definir los tipos
- Van a anotar los símbolos, van en el mismo paquete
- Aquí sencillo, podrían tener mas estructura

```
1 type Type struct {  
2     Id int  
3 }  
4 const (  
5     TUndef = iota  
6     TInt  
7     TFloat  
8     NTypes  
9 )
```

Ejemplo

```
10 var typeNames = []string{
11     TUndefined: "undef",
12     TInt:       "int",
13     TFloat:     "float",
14 }
15 var Types = []*Type{
16     TUndefined: &Type{TUndefined},
17     TInt:       &Type{TInt},
18     TFloat:     &Type{TFloat},
19 }
```

Ejemplo

- Hay que definir qué es compatible
- Y qué tipo devuelve
- El ejemplo que vemos tiene **tipado estricto o fuerte** (un símbolo, un tipo, incompatibles diferentes tipos, *strong typing*)
- Y variantes de **tipado débil** (castings automáticos. . . *weak typing*)
- Hay **tipado dinámico** (las asignaciones pueden cambiar el tipo de un símbolo, JavaScript, Python i.e. compilación vs ejecución) vs. **tipado estático**

Ejemplo

- En este ejemplo, tipado estricto, el tipo del símbolo devuelto es:
 - ▶ El del primer operando si todo va bien (que es el de todos)
 - ▶ TUndefined si ha habido un error
 - ▶ TUndefined es compatible con todos (para evitar errores en cascada)

Ejemplo

```
1 func (tp *Type) IsCompat(tp2 *Type, op int) bool {  
2     if tp == Types[TUndef] || tp2 == Types[TUndef] {  
3         return true  
4     }  
5     return tp.IsTypeCompat(tp2) && tp.IsCompatOp(op)  
6 }
```

Ejemplo

```
1 func (tp *Type) IsCompatOp(op int) bool {  
2     switch tp {  
3     case Types[TInt]:  
4         return strings.ContainsRune("+-*/%&|", rune(op))  
5     case Types[TFloat]:  
6         return strings.ContainsRune("+-*/", rune(op))  
7     default:  
8         return false  
9     }  
10 }
```

Ejemplo

```
1 func (tp *Type) IsTypeCompat(tp2 *Type) bool {  
2     return tp.Id == tp2.Id  
3 }  
4  
5 func (tp *Type) String() string {  
6     if tp == nil || tp.Id < TUndef || tp.Id >= NTypes {  
7         return "unktype"  
8     }  
9     return typeName[tp.Id]  
10 }
```


Ejemplo

```
1 stat      :      expr
2          {
3              if $1.DataType != sym.Types[sym.TUndef] {
4                  fmt.Printf( "%v\n", $1.SymValString() )
5              }else{
6                  fmt.Fprintf(os.Stderr, "Expression with unknown type\n")
7              }
8          }
9          |      .... el resto
```

Ejemplo

```
1  expr      :      '(' expr ')'  
2            { $$ = $2 }  
3            |      expr '+' expr  
4            {  
5                $$ = $1  
6                err := $.BinExpr($3, '+')  
7                if err != nil {  
8                    cl, _ := calclex.(*calcLex)  
9                    cl.Errorf(err.Error())  
10               }  
11           }  
12           |  
13           ....  
14           |      expr '%' expr  
15           {  
16               $$ = $1  
17               err := $.BinExpr($3, '%')  
18               if err != nil {  
19                   cl, _ := calclex.(*calcLex)  
20                   cl.Errorf(err.Error())  
21               }  
22           }  
23           |      ...
```

Ejemplo

- Importante: la gramática es lo más *sencilla* posible
- En el ejemplo de arriba, podría distinguir entre expresiones enteras y float (anotando símbolos The Lexer Hack) → *mala idea*
- Los tipos son comprobaciones semánticas que se hacen después de construir el AST, no van en la gramática (a menos que sea más sencillo, y aun así)

Ejemplo

```
1 func (s *Sym) BinExpr(s2 *Sym, op int) (err error) {
2     if !s.DataType.IsCompat(s2.DataType, op) {
3         errs := fmt.Sprintf("\tUncompat Types %s %s for op %c",
4             s.DataType, s2.DataType, rune(op))
5         err = errors.New(errs)
6         s.DataType = Types[TUndef]
7     } else {
8         s.OpVal(s2, op)
9     }
10    return err
11 }
```

Ejemplo

```
1 func (s *Sym) OpVal(s2 *Sym, op int) error {
2     if s2 == nil || s2.DataType == Types[TUndef] {
3         s.DataType = Types[TUndef]
4     }
5     switch s.DataType {
6     case Types[TInt]:
7         s.IntOp(s2, op)
8     case Types[TFloat]:
9         s.FloatOp(s2, op)
10    case Types[TUndef]:
11        return nil
12    default:
13        return errors.New("bad op")
14    }
15    return nil
16 }
```

Ejemplo

```
1 func (s *Sym) FloatOp(s2 *Sym, op int) error {
2     v := s2.FloatVal
3     if s2 == nil {
4         v = s.FloatVal
5         s.FloatVal = 0.0    //hack for UMINUS
6     }
7     switch op {
8     case '+':
9         s.FloatVal += v
10    case '-':
11        s.FloatVal -= v
12    case '*':
13        s.FloatVal *= v
14    case '/':
15        s.FloatVal /= v
16    default:
17        return errors.New("undef float op")
18    }
19    return nil
20 }
```

Tipos definidos por el usuario

- Van en la misma tabla (pero hay que añadir NewType y demás)
- En las acciones asociadas a las declaraciones de tipos se crean nuevos tipos y se meten en la tabla
- Tengo que ver si son compatibles (literales, tipos diferentes pero estructuralmente iguales, agregados, duck typing, constantes universales)
- Básicamente cuando tengo dos símbolos $x=y$ o $x+y$ tengo que ver si son compatibles

Tipos definidos por el usuario: registros

- Ojo, record no es un tipo de datos son N (uno por cada tipo de record definido por el usuario)
- Puedo hacer que si el tipo de dato concreto no es igual, no es compatible (**tipado nominal**)
- O puedo hacerlo estructuralmente compatible, **tipado estructural** (tuplas, inicializadores)

Ejemplo: registros, compatibilidad estructural

- El lenguaje me permite asignar dos records si tienen campos de los mismos tipos
- O lo puedo hacer sólo para inicializadores

Ejemplo: registros

- Ojo, los tipos de los campos pueden ser records también (llamada recursiva)

```
1 type Type struct {
2     Id int
3     isRecord bool
4     fld []*Type
5 }
6 func (tp *Type) IsTypeCompat(tp2 *Type) bool {
7     if tp.isRecord && tp2.isRecord {
8         if len(tp.fld) != len(tp2.fld) {
9             return false
10        }
11        for i, t := range tp.fld {
12            if !t.IsTypeCompat(tp2.fld[i]) {
13                return false
14            }
15        }
16        return true
17    }
18    return tp.Id == tp2.Id //si no hay alias: tp == tp2
19                          //si hay alias tp.derefType() == tp2.derefType()
20 }
```

Ejemplo: registros

- Puedo obligar a que los campos se llamen igual (lo normal)
- Para eso, tengo que tener un array de símbolos (de tipo campo) en el tipo

Ejemplo: registros, comprobación estándar

- Ojo, los tipos de los campos pueden ser records también (llamada recursiva)

```
1 type Type struct {
2     Id int
3     isRecord bool
4     fld []*Sym
5 }
6 func (tp *Type) IsTypeCompat(tp2 *Type) bool {
7     if tp.isRecord && tp2.isRecord {
8         if len(tp.fld) != len(tp2.fld) {
9             return false
10        }
11        for i, f := range tp.fld {
12            f2 := tp2.fld[i]
13            if f.name != f2.name {
14                return false
15            }
16            if !f.DataType.IsTypeCompat(f2.DataType) {
17                return false
18            }
19        }
20        return true
21    }
22    return tp.Id == tp2.Id //si no hay alias: tp == tp2
23                          //si hay alias tp.derefType() == tp2.derefType()
24 }
```

Tipos definidos por el usuario: arrays

- Más fácil, un sólo tipo, el de todos los elementos
- En algunos lenguajes el tamaño forma parte del tipo (Picky, Pascal, Ada)
- En otros no (C, Java)
- En Go sí, pero lo compensan las slices (más expresividad), en las que no
- Los strings suelen ser como los arrays, pero con predefinidos

Punteros

- Igual que con el resto, pueden ser tipos nominales (tienen un nombre y son incompatibles: Picky, ADA)
- O comparación estructural (o tipos anónimos, C)
- En otras palabras:
 - ▶ Tengo que darle nombre a `int*`
 - ▶ ¿Son compatibles todos los punteros a `int`?

Atravesar referencia de tipos

- Con expresiones como `x[3]` o `c.lastname`
- Al atravesar el tipo para acceder a un campo tengo que calcular un tipo a partir de otro
- Es sencillo:
 - ▶ Para arrays, es el tipo de los elementos (anotado en el tipo)
 - ▶ Para registros, es el tipo del campo, a partir del símbolo obtenido del nombre (se puede tener una hash de nombre → símbolo en el tipo)
 - ▶ Para punteros, cada vez que se atraviesa se quita un asterisco por ejemplo si atravieso `**int` → `*int` o si son tipos con nombre, se pone sencillamente un puntero al nombre de lo que apunta

Tipos definidos por el usuario: enumerados

- Conjunto de valores con nombre
- Array de nombres, valor máximo y mínimo
- Se suelen representar internamente como enteros (pero pueden ser incompatibles entre sí y con los enteros, nominales o estructurales)
- Un ejemplo los bool (si los hay)

Literales

- El valor 3 en Go es compatible con `float64` `int...` (constantes universales)
- En otros lenguajes no (por ejemplo C)
- En muchos lenguajes `nil` es universal, compatible con cualquier tipo (en C el tipo `void*`, de hecho, `NULL` es `(void *)0`)
- Alias de tipos: `rune` es un alias de `int`, son el mismo tipo con diferentes nombres pero son compatibles
- Al final es cambiar las funciones que comprueban compatibilidad
- En algunos lenguajes (C) hay inicializadores para todos los records, independientemente del tipo (tipo universal de agregado compatible si es estructuralmente igual)

```
1 struct point p = { 1, 2 };
```

Funciones/Métodos

- Las funciones/métodos tienen tipos (tipo del receptor, de los parámetros y de los valores de vuelta, similar a records)
- Importante para las llamadas a función, variables de función, punteros a función. . .
- Y para tipos de objetos (clases) e interfaces
- Métodos: normalmente funciones con parámetros extra especiales (el objeto sobre el que actúan o tipo receptor)

Duck typing

- Si tiene el mismo API es compatible (tipado estructural, pero en tiempo de ejecución)
- Uso de interfaces en Go (es ambos, estructural y **duck typing**)
- Por cierto: un interfaz es un conjunto de tipos de métodos
- Similar a un record, un array de tipos de métodos, pero en el runtime

Inferencia de tipos

- Igual que en las declaraciones se pone el símbolo del tipo
- Puede no declararse explícitamente, pero es similar
- Por ejemplo en Go: $x := \text{math.Sin}(z)$

Inferencia de tipos

- En algunos, lógicos (Prolog) o funcionales (Haskell) hay cosas más sofisticadas, como deducir el tipo de una función de la definición
- Requieren algoritmos de unificación (escribir ecuaciones de restricciones sobre los tipos, resolverlas)
- <https://eli.thegreenplace.net/2018/type-inference/>
<https://eli.thegreenplace.net/2018/unification/>
- *An Efficient Unification algorithm* Martelli, Montanari
- *Unification: A Multidisciplinary survey* Kevin Knight

Comprobación de tipos

- Variables y constantes
- Expresiones
- Funciones
- Objetos
- Polimorfismo
- Genéricos/Templates, más complicado, hay varias pasadas

Polimorfismo

- Cuando algo es compatible con tipos diferentes
- Ejemplo: nil en Go, es de tipo slice (*Ad hoc polymorphism*)
- Ejemplo: Constantes universales en Go (*Ad hoc polymorphism*)
- Ejemplo: Herencia, el padre es compatible con el hijo, puedo pasarle un camión si recibe un vehículo (*Subtyping polymorphism*)
- Ejemplo: Interfaces en Go (*Subtyping polymorphism*)
- Ejemplo: Métodos en Java puede haber con diferentes parámetros, se escoje el adecuado (*Parametric polymorphism*)

Polimorfismo paramétrico

- Ejemplo: Métodos en Java puede haber con diferentes parámetros, se escoje el adecuado (Parametric polymorphism)
- Ojo: en **tipado estático**, el valor de vuelta tiene que ser el mismo (o un supertipo o tipo compatible)

- Para poder hacer

```
int restos=g.Triturar(papel);
```

independientemente de lo que reciba la trituradora. El tipo de restos está definido.

Jerarquía de tipos (objetos, funcional)

- Requiere un grafo de dependencia de tipos (Subtyping, pero en árbol)
- En los lenguajes de orientación a objetos, hay parte en compilación y parte en ejecución (estática y dinámica)
- Ejemplo dinámico: **late binding**, **dispatching** basado en tablas por tipo. . .
- Complica el sistema de tipos (problema del diamante, implementar interfaces vs. herencia. . .)

Genéricos/Templates

- Problema interesante de solución difícil
- Si tengo tipado estricto, es difícil hacer estructuras de datos con implementaciones genéricas... (por ejemplo listas)
- ...que permitan meter cualquier tipo pero con comprobaciones de tipo para una concreta (se declara)
- Hay tres soluciones (cada una tiene sus usos dependiendo de los compromisos de ingeniería, eficiencia, el lenguaje, la complejidad del tipo...):
 - ▶ Tener una implementación para cada tipo (lista de enteros, lista de strings...), si el tipo es complicado, es poco DRY y fácil terminar con bugs
 - ▶ Saltarse el sistema de tipos (por ejemplo `void*` en C o `interface{}` en Go) → peligroso
 - ▶ Genéricos, templates o tipos paramétricos (puede utilizar en su implementación una de las anteriores generada)

Genéricos/Templates/Tipos paramétricos

- Idea: defino el tipo (por ejemplo lista) sobre un tipo de datos que pasaré como parámetro en la declaración

- Por ejemplo (java)

```
public class List <T>
{
    private ListMember <T> first;
    ...
}
```

- Luego en la declaración de variable digo de qué tipo es (por ejemplo Double):

```
List <Double> numbers = new List <Double> ();
```

- Tengo las ventajas de la protección de tipos y no tengo que implementar uno por cada parámetro

Tipos paramétricos: problemas

- Puede llegar a ser muy complicado de entender, si tengo varios parámetros y restricciones (el tipo parámetro tiene que cumplir un interfaz, o heredar o...)
- Básicamente hay dos formas principales de implementarlo y ambas son problemáticas:
 - ▶ N copias: una implementación para cada tipo (C++)
 - ▶ Borrado de tipos *Type erasure* (java)

Tipos paramétricos: type erasure

- Se comprueba el tipo en el uso (insertando type casts y demás, algunos suceden en tiempo de ejecución)
- Se borra toda referencia a tipo en los métodos y variables del tipo genérico (son object)
- Sólo hay una implementación de librería sobre object, y se ha perdido el tipo original
- C# arregla alguno de los problemas que salen de esto anotando con un tipo diferente cada instancia, que java no hace, porque no querían cambiar la VM
- La solución de C# añade coste en tiempo de ejecución (una de las cosas que se supone hacía buenos los genéricos)

Tipos paramétricos: N copias

- Básicamente genero N copias del genérico (o al menos de la parte genérica, o de los que se usan, se pueden hacer algunas optimizaciones)
- Problema: los programas y los librerías engordan (con explosión combinatoria si no tengo cuidado), caché de texto llena. . .

Pasadas del compilador

- Las comprobaciones semánticas se puede hacer en una pasada o en varias, como el resto de cosas
- Es, básicamente recorrerse varias veces el AST anotándolo, reescribiéndolo, realizando acciones. . .
- Construir el AST, anotar tipos, comprobar tipos. . .

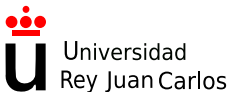
Tema 9: Generación de código

Autómatas y desarrollo avanzado de software

Gorka Guardiola Múzquiz

GSYC

23 de noviembre de 2022



Compilador sencillo, avanzado, intérprete

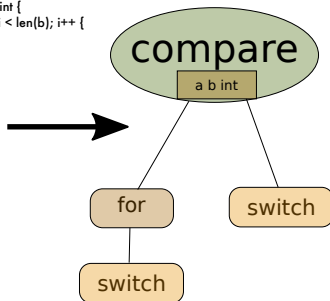
- Un compilador sencillo genera directamente código a partir del AST (máquina binario, ensamblador, otro lenguaje, etc.)
- Un compilador optimizador, normalmente *frontend*, *backend*
 - ▶ El *frontend* genera código intermedio (IR) del AST (puede haber optimización)
 - ▶ El *backend* lee el IR, optimiza, genera código
- Un intérprete ejecuta el código (puede o no construir el AST y/o generar código, ej. JIT)
- Síntesis (HDL)

Compilador sencillo, ejemplo con generación de código

- Ejemplo sencillo de un compilador
- Construye el AST, genera código

Compilador sencillo (traspas tema introducción)

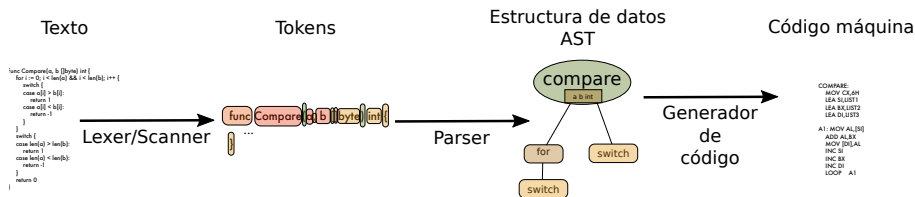
```
func Compare(a, b []byte) int {  
    for i := 0; i < len(a) && i < len(b); i++ {  
        switch {  
        case a[i] > b[i]:  
            return 1  
        case a[i] < b[i]:  
            return -1  
        }  
    }  
    switch {  
    case len(a) > len(b):  
        return 1  
    case len(a) < len(b):  
        return -1  
    }  
    return 0  
}
```



COMPARE:
MOV CX,6H
LEA SI,LIST1
LEA BX,LIST2
LEA DI,LIST3

A1: MOV AL,[SI]
ADD AL,BX
MOV [DI],AL
INC SI
INC BX
INC DI
LOOP A1

Compilador sencillo (traspas tema introducción)



Compilador sencillo, ejemplo con generación de código

Ejemplo de lenguaje de entrada, llaves programa principal, expresiones, asignaciones/declaraciones, bucles

```
{  
    z = 2  
    for(3, 4*7+z){  
        x=3.0  
        (x *3.0)+5.0  
        3.0+ 2.0  
    }  
}
```

Compilador sencillo, ejemplo con generación de código

Ejemplo de lenguaje de salida, líneas numeradas, GOTO, GOTO condicional, asignación, expresiones

```
0: PROG
1: PUSH VAR z
2: z = 2
3: (3)>(2 + 7 * 4) GOTO 10
4: PUSH VAR x
5: 7
6: x = 3.000000
7: 5.000000 + 3.000000 * 3.000000
8: 2.000000 + 3.000000
9: GOTO 3
10: POP VAR x
11: 2.000000
12: POP VAR z
13: END
```

Compilador sencillo, ejemplo con generación de código

- Los símbolos en un paquete, generador y AST
- El lexer en otro paquete
- El compilador en otro paquete
- El main por separado

Compilador sencillo, ejemplo con generación de código

```
$ find .  
.  
./main.go  
./comp  
./comp/calc.y  
./comp/comp_test.go  
./lex  
./lex/lex.go  
./lex/lex_test.go  
./sym  
./sym/ast.go  
./sym/ast_test.go  
./sym/gen.go  
./sym/gen_test.go  
./sym/sym.go  
./sym/sym_test.go  
./sym/type.go  
./sym/type_test.go
```


Generación de código, variables

- Algunos símbolos tienen reflejo en la memoria
- Variables, registros, arrays. . .
- Se les pueden asignar cosas, sacar su dirección en memoria, etc.
- Se les suele llamar lvalues (Left-values, lado izquierdo de una asignación)

Generación de código, valores

- Las expresiones evalúan, es decir se puede obtener un valor de ellas
- Pueden estar a la derecha de una asignación
- Se les suele llamar rvalues (Right-values, lado derecho de una asignación)

Generación de código, constantes

- Las constantes son conceptos de compilación, no de generación de código
- Pueden no existir (por ejemplo tengo definido una constante $A = 4$
- y aparece en $c = 3 + A$
- Se puede generar $c = 7$ y ya
- Puede ser importante para depurar que se generen símbolos en el fichero (o no)

Compilador sencillo, ejemplo con generación de código

Símbolos

```
1 const (  
2     SNone    = iota  
3     SKey     // Keyword  
4     SProg    // the program  
5     SConst   // constant or literal  
6     SVar     // object definition  
7     SUnary   // unary expression  
8     SBinary  // binary expression  
9     SIter    // loop  
10    SAssign  
11 )
```

Compilador sencillo, ejemplo con generación de código

Símbolos

```
1 type Sym struct {  
2     name      string  //name of the var, literal, etc.  
3     sType     int      //SKey, SConst...  
4  
5     DataType *Type  
6  
7     lex.Place //file and line no  
8     //other stuff for the tree  
9  
10    /* one of */  
11    FloatVal float64  
12    IntVal  int64  
13    Expr    *Expr  
14    Prog    *Prog  
15    Asign   *Asign  
16    lter     *lter  
17    val     *Sym  /* for var when eval */  
18 }
```

Compilador sencillo, ejemplo con generación de código

AST

```
1 type Expr struct {  
2     Left *Sym  
3     Right *Sym  
4     Op int  
5     Val *Sym  
6 }  
7 type Prog struct {  
8     b *Body  
9 }  
10 type Assign struct {  
11     LVal *Sym  
12     RVal *Sym  
13 }
```

Compilador sencillo, ejemplo con generación de código

AST

```
1 type Iter struct {  
2     St *Sym  
3     End *Sym  
4     b *Body  
5 }  
6 type Body struct {  
7     lvl int      /* for printing */  
8     env Env  
9     sent []*Sym /* iter expr asig */  
10 }
```

Compilador sencillo, ejemplo con generación de código

AST

```
1 func (envs *StkEnv)NewExpr(el *Sym, er *Sym, op int, val *Sym, place lex.Place) (s *Sym){
2     var err error
3     if op == 0 {
4         panic("bad expr")
5     }
6     if er == nil {
7         s, err = envs.NewSym("", SUnary)
8         if err != nil {
9             panic("creating unary exp sym")
10        }
11    } else {
12        s, err = envs.NewSym("", SBinary)
13        if err != nil {
14            panic("creating binary exp sym")
15        }
16    }
17    s.Expr = &Expr{er, el, op, val}
18    s.Place = place
19    return s
20 }
```


Compilador sencillo, ejemplo con generación de código

AST

```
1 const ProgName = "PROG:::P"
2 func (envs *StkEnv) NewProg(b *Body, place lex.Place) (s *Sym){
3     s, err := envs.NewSym(ProgName, SProg)
4     if err != nil {
5         panic("creating prog sym"+err.Error())
6     }
7     s.Prog = &Prog{b}
8     s.Place = place
9     return s
10 }
```

Compilador sencillo, ejemplo con generación de código

AST

```
1 func NewIter(stExp *Sym, endExp *Sym, body *Body,  
2                                     place lex.Place) (s *Sym){  
3     s, err := NewGSym("", SIter)  
4     if err != nil {  
5         panic("creating iter sym")  
6     }  
7     s.Iter = &Iter{stExp, endExp, body}  
8     s.Place = place  
9     return s  
10 }
```

Compilador sencillo, ejemplo con generación de código

AST

```
1 func (envs *StkEnv) NewConst(typeC lex.TokType,
2                               place lex.Place)(s *Sym, err error){
3     s, err = envs.NewSym("", SConst)
4     if err != nil {
5         fmt.Fprintf(os.Stderr, "bad const")
6         return nil, err
7     }
8     switch typeC {
9     case lex.TokFloatVal:
10        s.DataType = Types[TFloat]
11     case lex.TokIntVal:
12        s.DataType = Types[TInt]
13     default:
14        fmt.Fprintf(os.Stderr, "bad const")
15        return nil, err
16     }
17     s.Place = place
18     return s, nil
19 }
```

Compilador sencillo, ejemplo con generación de código

AST

```
1 func (b *Body) AddSent(s *Sym){  
2     b.sent = append(b.sent, s)  
3 }  
4  
5 func NewBody(lvl int, env Env)(b *Body){  
6     return &Body{lvl: lvl, env: env}  
7 }
```

Compilador sencillo, ejemplo con generación de código

Símbolos

```
1 func NewGSym(name string, sType int) (s *Sym, err error) {
2     return &Sym{sType: sType}, nil
3 }
4 func NewAsig(lVal *Sym, rVal *Sym) (s *Sym){
5     s, err := NewGSym("", SAsign)
6     if err != nil {
7         panic("creating asig")
8     }
9     s.Asign = &Asign{lVal, rVal}
10    return s
11 }
```

Compilador sencillo, ejemplo con generación de código

Símbolos

```
1 func (envs *StkEnv)NewSym(name string, sType int) (s *Sym, err error) {
2     s = &Sym{sType: sType, name: name}
3     if sType == SKey {
4         e := envs.e[0]
5         e[name] = s
6         return s, nil
7     }
8     if name == "" { //unnamed constants, I make up the name
9         name = fmt.Sprintf(":%d", envs.idGen)
10        envs.idGen++
11    }
12    s.name = name
13    e := envs.e[len(envs.e)-1]
14    if _, ok := e[name]; ok {
15        return nil, errors.New("already declared sym: " + name)
16    }
17
18    e[name] = s
19    return s, nil
20 }
```

Compilador sencillo, ejemplo con generación de código

Símbolos

```
1 func (envs *StkEnv) GetSym(name string) (s *Sym) {
2     for i := len(envs.e) - 1; i >= 0; i-- {
3         if s, ok := envs.e[i][name]; ok {
4             return s
5         }
6     }
7     return nil
8 }
9
10 func (s *Sym) SetVal(s2 *Sym) {
11     s.val = s2
12 }
```

Compilador sencillo, ejemplo con generación de código

Dos tipos, int y float. Anotación de tipos en el AST

```
1 func (s *Sym) Annotate() {  
2     switch s.sType {  
3     case SProg:  
4         for _, v := range s.Prog.b.sent {  
5             v.Annotate()  
6         }  
7     case SConst:  
8         return  
9     case SVar:  
10        //inferred type from expression  
11        s.DataType = s.val.DataType
```


Compilador sencillo, ejemplo con generación de código

Dos tipos, int y float. Anotación de tipos en el AST

```
12  case SAssign:
13      s.Assign.RVal.Annotate()
14      if s.Assign.LVal.DataType != nil {
15          lv := s.Assign.LVal
16          rv := s.Assign.RVal
17          allID := lv.DataType != Types[TUndef]
18          allID = allID && rv.DataType != Types[TUndef]
19          if lv.DataType.IsTypeCompat(rv.DataType) && allID {
20              lv.DataType = rv.DataType
21          } else {
22              lv.DataType = Types[TUndef]
23          }
24      }
```

Compilador sencillo, ejemplo con generación de código

Dos tipos, int y float. Anotación de tipos en el AST

```
1  case SBinary :
2      le := s.Expr.Left
3      re := s.Expr.Right
4      le.Annotate()
5      re.Annotate()
6      isUR := re.DataType == Types[TUndef]
7      isUL := re.DataType == Types[TUndef]
8      anyU := isUR || isUL
9      isC := le.DataType.IsCompat(re.DataType, s.Expr.Op)
10     s.DataType = Types[TUndef]
11     if isC && !anyU {
12         s.DataType = le.DataType
13     }
14 case SUnary :
15     le := s.Expr.Left
16     le.Annotate()
17     isUL := re.DataType == Types[TUndef]
18     if le.DataType.IsCompat(nil, s.Expr.Op) && !isUL {
19         s.DataType = le.DataType
20     }
```

Compilador sencillo, ejemplo con generación de código

Dos tipos, int y float. Anotación de tipos en el AST

```
1  case SIter :  
2      s.Iter.St.Annotate()  
3      s.Iter.End.Annotate()  
4      for _, v := range s.Iter.b.sent {  
5          v.Annotate()  
6      }  
7  }
```

Compilador sencillo, ejemplo con generación de código

El parser de yacc

```
1 %union {
2     node *sym.Sym
3     body *sym.Body
4     name string
5 }
6 %type <node> expr
7 %type <node> prog
8 %type <body> body
9 %type <node> stat
10
11 %token <node> NUMBER
12 %token <name> VAR
13 %token <node> ITER
14
15 %left '|'
16 %left '&'
17 %left '+' '-'
18 %left '*' '/' '%'
19 %left UMINUS
```

Compilador sencillo, ejemplo con generación de código

El parser de yacc

```
1 prog      : '{' body
2           {
3             cl, _ := calclex.(*calcLex)
4             $$ = cl.parseSt.stkEnv.NewProg($2, cl.Place)
5           } '}' {
6             cl, _ := calclex.(*calcLex)
7             prog := cl.parseSt.stkEnv.GetSym(sym.ProgName);
8             if cl.nErr == 0 && prog != nil{
9               c:= &sym.Code{}
10              prog.Gen(c) //TODO, pinchar c en el parser para retornarlo
11              fmt.Printf("%s", c)
12            }
13            cl.parseSt.stkEnv.PopEnv()
14          }
```

Compilador sencillo, ejemplo con generación de código

El parser de yacc

```
1 body : /* empty */
2       {
3           cl, _ := calclex.(*calcLex)
4           cl.parseSt.stkEnv.PushEnv()
5           lvl := cl.parseSt.stkEnv.CurrLevel()
6           $$ = sym.NewBody(lvl, cl.parseSt.stkEnv.CurrEnv())
7       }
8 | body stat
9   {
10      if $1 != nil {
11          //add line
12          $$ = $1
13          $$ . AddSent($2)
14      } else {
15          panic("nil body should not happen")
16      }
17  }
```

Compilador sencillo, ejemplo con generación de código

El parser de yacc

```
1 | error '}'  
2 | {  
3 |     cl, _ := calclex.(*calcLex)  
4 |     cl.nErr++  
5 |     $$ = nil    //TODO, pichar un body retocado para error  
6 |     Errflag = 0;  
7 | }  
8 ;
```

Compilador sencillo, ejemplo con generación de código

El parser de yacc

```
1      |      ITER '(' expr ',' expr ')' '{' body
2      {
3          cl, _ := calclex.(*calcLex)
4          *($1) = *sym.NewIter($3, $5, $8, cl.Place)
5          $$ = $1
6      } '}' {
7          cl, _ := calclex.(*calcLex)
8          cl.parseSt.stkEnv.PopEnv()
9      }
10     | '\n' {
11         dprintf("empty line\n")
12     }
13     ;
```


Compilador sencillo, ejemplo con generación de código

El parser de yacc

```
1 stat      :      '(' expr ')'  
2          {  
3              dprintf("expr: ( expr )\n")  
4              $$ = $2  
5          }  
6          | expr '+' expr  
7          {  
8              dprintf("expr: expr + expr\n")  
9              cl, _ := calclex.(*calcLex)  
10             pS := cl.parseSt  
11             $$ = pS.stkEnv.NewExpr($1, $3, '+', nil, cl.Place)  
12         }  
13         | ...  
14         | '-' expr      %prec UMINUS  
15         {  
16             cl, _ := calclex.(*calcLex)  
17             pS := cl.parseSt  
18             $$ = pS.stkEnv.NewExpr($2, nil, '-', nil, cl.Place)  
19             $$ . OpVal(nil, '-')  
20         }
```

Compilador sencillo, ejemplo con generación de código

El parser de yacc

```
1 stat      :    expr '\n'
2          {
3              cl, _ := calclex.(*calcLex)
4              $1.Annotate()
5              if $1.DataType != sym.Types[sym.TUndef] {
6                  fmt.Fprintf(os.Stderr, "Expr: unknown type\n")
7                  cl.nErr++
8              }
9              $$ = $1
10         }
```

Compilador sencillo, ejemplo con generación de código

El parser de yacc

```
1 |   VAR '=' expr '\n'
2 |   {
3 |       cl, _ := calclex.(*calcLex)
4 |       s := cl.parseSt.stkEnv.GetSym($1);
5 |       var err error
6 |       if s != nil && s.SymType() != sym.SVar {
7 |           fmt.Fprintf(os.Stderr, "Assign to %s, not lval\n", $1)
8 |           cl.nErr++
9 |           s = nil
10 |      }
11 |      if s == nil {
12 |          s, err = cl.parseSt.stkEnv.NewSym($1, sym.SVar)
13 |          if err != nil {
14 |              panic(err)
15 |          }
16 |      }
17 |      s.Place = cl.Place
18 |      s.DataType = $3.DataType
19 |      s.SetVal($3)
20 |      $$ = sym.NewAsig(s, $3, cl.Place)
21 |  }
```

Compilador sencillo, ejemplo con generación de código

El parser de yacc

```
1 func (cl *calcLex) Lex(lval *calcSymType) int {
2     tok, err := cl.Scan()
3     if err != nil {
4         panic(err)
5     }
6     pS := cl.parseSt
7     switch tok.Type {
8     case lex.TokFloatVal:
9         lval.node, _ = pS.stkEnv.NewConst(tok.Type, cl.Place)
10        lval.node.FloatVal = tok.FloatVal
11        return NUMBER
12    case lex.TokIntVal:
13        lval.node, _ = pS.stkEnv.NewConst(tok.Type, cl.Place)
14        lval.node.IntVal = tok.IntVal
15        return NUMBER
16    }
17 }
```

Compilador sencillo, ejemplo con generación de código

El parser de yacc

```
1  case lex.TokId:
2      if keyW:= sym.NewKey(tok.Lexema, cl.Place); keyW != nil{
3          lval.node, _ = kewW
4          return ITER
5      }
6      lval.name = tok.Lexema
7      return VAR
8  case lex.TokEof:
9      return -1
10 case lex.TokBad:
11     panic("error, bad token")
12     return -1
13 default:
14     return int(tok.Type)    //trick
15 }
16 }
```

Compilador sencillo, ejemplo con generación de código

Generación de código: *Backpatching*

```
1 type Code struct {
2     lstk LabelStack
3     line []string
4 }
5
6 type LabelStack struct {
7     labels []int //line to change, it is a goto
8 }
9
10 func (lstk *LabelStack) pushLabel(line int) {
11     lstk.labels = append(lstk.labels, line)
12 }
13
14 func (lstk *LabelStack) popLabel() (line int) {
15     l := len(lstk.labels)
16     line = lstk.labels[l-1]
17     lstk.labels = lstk.labels[:l-1]
18     return line
19 }
```

Compilador sencillo, ejemplo con generación de código

Generación de código

```
1 func (s *Sym) GenExpr() string {
2     if s == nil {
3         return "nil"
4     }
5     str := ""
6     switch s.sType {
7     case SVar:
8         str += s.name
9     case SAsign:
10        str += "Asig:" + s.Asign.LVal.GenExpr() + " = "
11        str += s.Asign.RVal.GenExpr()
12    case SBinary:
13        str += s.Expr.Left.GenExpr()
14        str += " " + fmt.Sprintf("%c", rune(s.Expr.Op))
15        str += " " + s.Expr.Right.GenExpr()
16    case SUnary:
17        str += s.Expr.Left.GenExpr()
18        str += " " + fmt.Sprintf("%c", rune(s.Expr.Op))
```

Compilador sencillo, ejemplo con generación de código

Generación de código

```
1  case SConst:
2      switch s.DataType {
3      case Types[TInt]:
4          str += fmt.Sprintf("%d", s.IntVal)
5      case Types[TFloat]:
6          str += fmt.Sprintf("%f", s.FloatVal)
7      default:
8          str += "?"
9      }
10 case SProg:
11     fallthrough
12 case SIter:
13     fallthrough
14 default:
15     panic("should not happen, genExpr not an expr")
16 }
17 return str
```


Compilador sencillo, ejemplo con generación de código

Generación de código

```
1 func (b *Body) GenVars(c *Code, op string) {
2     env := b.env
3     for _, v := range env {
4         if v.sType == SVar {
5             l := len(c.line)
6             line := fmt.Sprintf("%d: %s VAR %s ", l, op, v.name)
7             c.line = append(c.line, line)
8         }
9     }
10 }
11 func (b *Body) GenBody(c *Code) {
12     for _, v := range b.sent {
13         v.Gen(c)
14     }
15 }
```

Compilador sencillo, ejemplo con generación de código

Generación de código

```
1 func (s *Sym) Gen(c *Code) {
2     if s == nil {
3         return
4     }
5     nl := len(c.line)
6     pref := fmt.Sprintf("%d: ", nl)
7     line := pref
8     switch s.sType {
9     case SProg:
10         c.line = append(c.line, pref+"PROG")
11         b := s.Prog.b
12         b.GenVars(c, "PUSH")
13         b.GenBody(c)
14         b.GenVars(c, "POP")
15         line = fmt.Sprintf("%d: END", len(c.line))
16         c.line = append(c.line, line)
```

Compilador sencillo, ejemplo con generación de código

Generación de código

```
1  case SConst:
2      switch s.DataType {
3      case Types[TInt]:
4          line += fmt.Sprintf("%d", s.IntVal)
5      case Types[TFloat]:
6          line += fmt.Sprintf("%f", s.FloatVal)
7      default:
8          panic("bad const, should be detected with Undef")
9      }
10     c.line = append(c.line, line)
11 case SVar:
12     line += s.GenExpr()
13     c.line = append(c.line, line)
```

Compilador sencillo, ejemplo con generación de código

Generación de código

```
1  case SAssign:
2      line += s.Assign.LVal.name
3      line += " = " + s.Assign.RVal.GenExpr()
4      c.line = append(c.line, line)
5  case SBinary:
6      line += s.GenExpr()
7      c.line = append(c.line, line)
8  case SUnary:
9      line += s.GenExpr()
10     c.line = append(c.line, line)
```

Compilador sencillo, ejemplo con generación de código

Generación de código

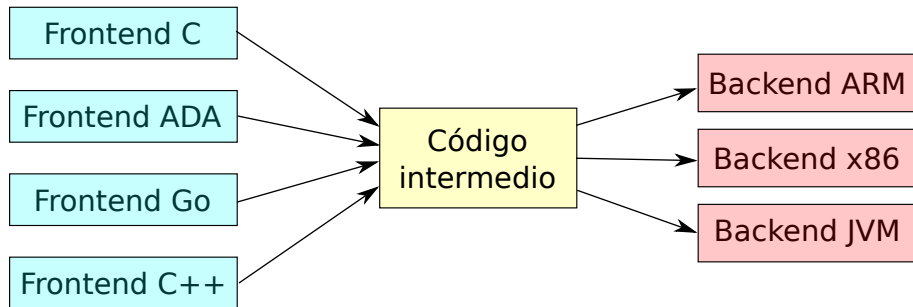
```
1  case SIter:
2      pref = fmt.Sprintf("%d: ", len(c.line))
3      line = pref
4
5      c.lstk.pushLabel(nl)
6      line += "(" + s.Iter.St.GenExpr() + ")" + ">"
7      line += s.Iter.End.GenExpr() + " GOTO"
8      c.line = append(c.line, line)
9
10     b := s.Iter.b
11     b.GenVars(c, "PUSH")
12     b.GenBody(c)
13     gotoLine := c.lstk.popLabel()
14     line = fmt.Sprintf("%d: GOTO %d", len(c.line), gotoLine)
15     c.line = append(c.line, line)
16     c.line[gotoLine] += fmt.Sprintf(" %d", len(c.line))
17     b.GenVars(c, "POP")
18 default:
19     panic("should not happen, gen with unknown type")
20 }
21 return
22 }
```

Compilador sencillo, ejemplo con generación de código

Generación de código

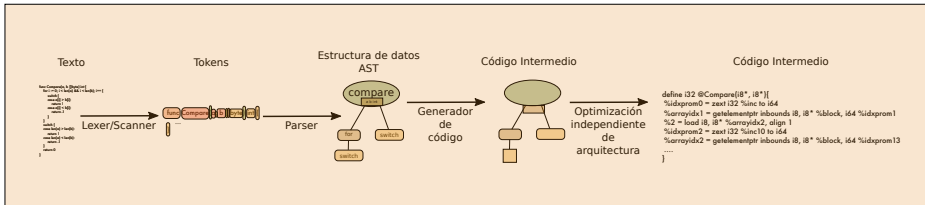
```
1 func (c *Code) String() string {  
2     str := ""  
3     for i := range c.line {  
4         str += c.line[i] + "\n"  
5     }  
6     return str  
7 }
```

Compilador optimizador, traspas intro

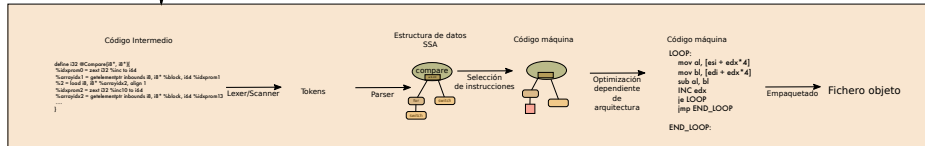


Compilador optimizador, traspas intro

FRONTEND

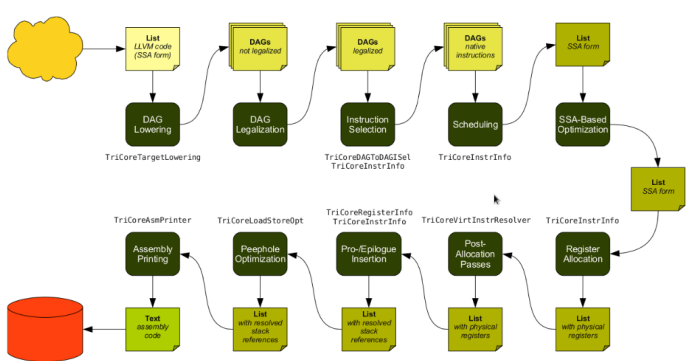


BACKEND



Compilador optimizador, traspas intro

Ejemplo: backend de llvm



Código intermedio, IR

- Representación fácil de optimizar y generar código
- Cada compilador la suya: polaca inversa (linealizada) TAC, SSA. . .
- **TAC** 3AC (Three address code), se escriben todas la operaciones $a = b \text{ OP } c$
- *Static Single Assignment Form* **SSA**, cada asignación crea una variable nueva que me invento
- Ambas útiles para las partes que hace el generador de código (fácil optimizar, asignar registros. . .)

Código intermedio, SSA

Cambia:

$$a = a + 1$$
$$a = 3 * a$$

Por:

$$a1 = a + 1$$
$$a2 = 3 * a1$$

Optimizaciones

- Independientes de arquitectura (frontend: frontend a IR, IR a IR)
- Dependientes de arquitectura (backend: IR a IR, IR a salida)

Optimizaciones

- Evaluación de constantes (*folding/propagation*) (reescritura del AST, representaciones intermedias)
- Reutilización de subexpresiones
- Inlining de funciones (sustituir una llamada por la función)
- Eliminación de código muerto, definido y no usado (análisis de flujo, *UD Chain, DU Chain*)
- **GVN** *Global Value Numbering* (elimina código redundante sobre una representación SSA)
- Strength reduction (sustitución de operaciones por otras más baratas, shift por multiplicación, por ejemplo)
- Optimización de última llamada (*tail call recursion*, convierte una llamada recursiva en un *JMP*)
- Superoptimización (probar formas para operaciones sin bucles, fuerza bruta)
- *Peephole optimization* (sustituir conjuntos de instrucciones por otras más cortas o quitar redundantes), reconoce un patrón en una ventana de código

Cosas que se deben hacer

- Asignación de registros, *Register allocation* (por ejemplo, con coloreado, *register coloring*)
- Selección de instrucciones, *Instruction Selection* (pensando en pipelining etc., orden de evaluación)
- Manejo de la pila (push y pops, etc. cambia si es *caller save* o *callee save*), manejo de memoria (GC, cuenta referencias. . .)

Técnicas importantes

- Backpatching (en el ejemplo anterior)
- Análisis de escape (Escape analysis) pila vs. heap

- AST
- Optimizaciones de alto nivel sobre el AST (constantes, inlining en algunos casos, reescritura de expresiones, last call. . .)
- Se genera el código intermedio IR: SSA/TAC (backpatching, asignación de memoria)
- Generación de la tabla de símbolos para el enlazado (no resueltos, provee)

Compilador, backend

- Se lee el el IR, se construye un DAG (*Directed Acyclic Graph*)
- Selección de las instrucciones
- Planificación de las instrucciones (scheduling, teniendo en cuenta pipeline, etc.)
- Optimizaciones SSA (GVN...)
- Asignación de los registros
- Generación de código para llamadas a función (*epilogue/prologue insertion*)
- *Peephole optimizations*
- Impresión ensamblador/escritura código máquina
- Sobre la marcha, se hace el rellenado de la tabla de símbolos para el enlazado (resueltos)

Intérprete

- El objetivo es ejecutar el código, no generarlo
- Desde no tener AST (algún ejemplo que hemos visto) hasta optimizador *JIT/hot code* hashing
- Con AST, similar a generar código
- Evaluando las expresiones y ejecutando el control de flujo
- Puedo tener una máquina abstracta/virtual o no. Si no la tengo puedo sencillamente usar el lenguaje
- Ejemplo de esto último, llego al iterador, y escribo un bucle en el lenguaje del intérprete

Síntesis

- Se describe el circuito con variables, if/then, etc. (RTL, *register transfer level*, Verilog, VHDL)
- Se construye el AST, se hacen las optimizaciones independientes de tecnología
- Se establece la correspondencia con la tecnología (*DAG covering*: ej. árbol de nands, librerías expresadas con eso, recubrir el DAG), generación de netlist
- Posicionamiento y encaminamiento (*placing and routing*)
- Generación de la máscara
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-884-complex-digital-systems-spring-2005/lecture-notes/105_synthesis.pdf
- <https://people.eecs.berkeley.edu/~keutzer/classes/244fa2004/pdf/7-1-techmap.pdf>

Síntesis



Síntesis, RTL $\rightarrow$ Lógica

- Se construye el AST
- Se genera un grafo de componentes (Registros, lógica, sumadores. . .)

Síntesis, Optimizaciones independientes de tecnología

- No del todo independientes
- Pero en la medida de lo posible (sencillo, general, *frontend* vs. *backend*)
- Minimización booleana (Antes se hacía con mapas de Karnaugh a mano, misma idea),
- Se usa **Quine-McCluskey**
- Peor caso exponencial, se usan heurísticas

Síntesis, correspondencia con la tecnología

- DAG covering (Directed Acyclic Graph)
- Algoritmo para recubrir el netlist (en forma normal, Nands de dos entradas e inversores)
- Usa programación dinámica (memorizar subproblemas ya resueltos por el algoritmo) similar a selección de instrucciones

Síntesis, optimizaciones dependientes de la tecnología

- Otros componentes mas complejos (Más complejos, más lentos pero menos area)
- Load buffering (añadir inversores o buffers para arreglar problemas de latencias en el camino crítico, evitar espúreos)
- Cambiar el tamaño de transistores en el camino crítico (integrados, no FPGA)
- Cambiar Latches y registros para evitar latencias adicionales en el camino crítico
- Mejorar que sea posible el encaminamiento, evitar congestión, evitar zonas con muchos cables

Síntesis, posicionamiento y encaminamiento

- Suele ser jerárquico tanto desde el HDL como FPGAs como IC (*macro blocks, floor planning*)
- Diferente problema en FPGAs que en IC, en un caso seleccionar componentes e interconexiones, en el otro, generar máscara
- En muchos casos, aun así, IC se reduce a un problema similar, suponiendo celdas que interconecta (*sea-of-gates router*)
- <http://opencircuitdesign.com/qrouter/>

Síntesis: referencias

- Devadas, Srinivas, Abhijit Ghosh, and Kurt William Keutzer. "Logic synthesis." (1994), McGraw-Hill Series on Computer Engineering, ISBN-10: 0070165009
- C. Wolf, *Design and Implementation of the Yosys Open SYnthesis Suite*, Bachelor Thesis, Vienna University of Technology, 2013
- El código de Yosys, <https://github.com/YosysHQ/> y Arachne, <https://github.com/cseed/arachne-pnr>
- <https://github.com/YosysHQ/yosys/blob/master/manual/literature.bib>

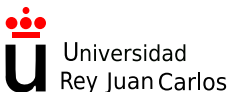
Tema 10: Implementación de un intérprete

Autómatas y desarrollo avanzado de software

Gorka Guardiola

GSYC

23 de noviembre de 2022



Hay varias maneras

- A partir del AST generar código para una máquina virtual/abstracta, de golpe (compilador) sobre la marcha (JIT)
- Ejecutar el AST directamente (tras construirlo o mientras se construye)
- Vamos a ver el segundo, que es fácil y diferente de un compilador

Ejecutar el AST

- El AST es el texto del programa (sólo lectura)
- Necesitamos memoria (principal y pila) y un flujo de ejecución (puntero de pila, contador de programa)
- Pero lo hacemos a alto nivel, con tipos de datos

Ejecutar el AST

- Memoria → Variables → Símbolos de ejecución
- Pila → Pila de símbolos de ejecución (puntero de pila)
- Contador de programa → Vamos recorriendo el AST

Símbolos

- Cuando una variable entra en ámbito, instanciamos un símbolo de ejecución (PUSH)
- Cuando sale de ámbito, lo borramos (POP)
- Estos símbolos tendrán un valor que es el valor en memoria de la variable
- Tendrá un valor (en el AST sólo hay valor para símbolos que representan constantes)
- Si soy cuidadoso puedo usar mi misma implementación de la tabla de símbolos (ojo)

Intérprete

- En un paquete separado
- O en el mismo de la tabla de símbolos
- En el ejemplo lo segundo, pero lo primero es mas limpio

Efectos laterales

- Aparte de ejecutar el programa
- Hay que realizar los efectos laterales (imprimir cosas. . .)

Ejemplo calculadora

- Calculadora sencilla que ya hemos visto con yacc
- Con un extra, si evaluamos una variable desnuda, imprime su valor (por ejemplo, línea 11)
- Ámbitos, bucles (con variable de control anónima), asignaciones y expresiones

```
1 {  
2     z = 2  
3     p = 4  
4     for (3, 4 * 7 + z) {  
5         x = 3.0  
6         (x * 3.0) + 5.0  
7         3.0 + 2.0  
8         p = p + 1  
9         p  
10    }  
11    z  
12 }
```

Ejemplo calculadora

- Infraestructura de depuración

```
1 const DEval = false
2
3 func (envs *StkEnv) dprintf(s string, v ...interface{}) {
4     lvl := len(envs.e)
5     indent := strings.Repeat("\t", lvl)
6     if DEval {
7         prefix := fmt.Sprintf("%sINTERP: ", indent)
8         fmt.Fprintf(os.Stderr, prefix+s, v...)
9     }
10 }
11 func prVars(envs *StkEnv) {
12     currEnv := envs.CurrEnv()
13     for _, v := range currEnv {
14         if v.sType == SVar {
15             envs.dprintf("VAR %s\n", v.EvalString())
16         }
17     }
18 }
```

Ejemplo calculadora

- Variables
- Copiar del ámbito lexico al de ejecución

```
1 func (b *Body) PushVars(envs *StkEnv) {
2     env := b.env
3     for _, v := range env {
4         if v.sType == SVar {
5             envs.dprintf("PUSHING VAR %s\n", v.name)
6             s, err := envs.NewSym(v.name, SVar)
7             if err != nil {
8                 panic(err)
9             }
10            s.DataType = v.DataType
11            val, _ := NewGSym("", SConst)
12            s.SetVal(val)
13        }
14    }
15 }
```

Ejemplo calculadora

- Copiar el valor de un símbolo (variable)

```
1 func (s *Sym) CopyValFrom(s2 *Sym) {  
2     s.DataType = s2.DataType  
3     switch s.DataType {  
4     case Types[TInt]:  
5         s.IntVal = s2.IntVal  
6     case Types[TFloat]:  
7         s.FloatVal = s2.FloatVal  
8     default:  
9         fmt.Fprintf(os.Stderr, "Copying unknown type")  
10    }  
11 }
```

Ejemplo calculadora

- Intérprete, ámbito global

```
1 func (s *Sym) Interp(envs *StkEnv) (val *Sym) {
2     if s.DataType == Types[TUndef] {
3         return
4     }
5     switch s.sType {
6     case SProg:
7         envs.dprintf("SProg\n")
8         b := s.Prog.b
9         envs.PushEnv()
10        b.PushVars(envs)
11        b.Interp(envs)
12        envs.PopEnv()
```

Ejemplo calculadora

• Intérprete

```
13  case SIter:
14      envs.dprintf(" SIter\n")
15      iter := s.lter
16      envs.PushEnv()
17      iter.b.PushVars(envs)
18      //I don't need an idx variable, because it is anonymous
19      start := s.lter.St.Eval(envs).IntVal
20      end := s.lter.End.Eval(envs).IntVal
21      envs.dprintf(" lter i:%d to %d\n", start, end)
22      for i := start; i < end; i++ {
23          envs.dprintf(" lter — i:%d to %d\n", start, end)
24          iter.b.Interp(envs)
25          end = s.lter.End.Eval(envs).IntVal
26      }
27      envs.PopEnv()
```

Ejemplo calculadora

• Intérprete

```
28     case SAssign :
29         envs.dprintf("SAssign\n")
30         valVar := s.Assign.RVal.Interp(envs)
31         v := envs.GetSym(s.Assign.LVal.name)
32         v.SetVal(valVar)
33         v.DataType = valVar.DataType
34         prVars(envs)
35     case SConst, SVar, SBinary, SUnary:
36         val = s.Eval(envs)
37     default :
38         panic("not executable")
39 }
40 return val
41 }
```


Ejemplo calculadora

- Interpretar un Body, secuencia de sentencias

```
42 func (b *Body) Interp(envs *StkEnv) {  
43     envs.dprintf("Body\n")  
44     for _, sent := range b.sent {  
45         if sent == nil {  
46             continue  
47         }  
48         val := sent.Interp(envs)  
49         if sent.sType == SVar {  
50             fmt.Printf("VAR %s %s\n", sent.name, val.EvalString())  
51         }  
52     }  
53 }
```

Ejemplo calculadora

- Imprimir el valor de una variable

```
53 func (s *Sym) EvalString() string {
54     if s == nil {
55         return "nil"
56     }
57     str := ""
58     switch s.sType {
59     case SVar:
60         str += s.name + "="
61         if s.val != nil {
62             str += s.val.EvalString()
63         }
64     case SConst:
65         switch s.DataType {
66         case Types[TInt]:
67             str += fmt.Sprintf("%d", s.IntVal)
68         case Types[TFloat]:
69             str += fmt.Sprintf("%f", s.FloatVal)
70         default:
71             str += "?"
72         }
73     default:
74         str = "Unknown"
75     }
76     return str
77 }
```

Ejemplo calculadora

- Evaluación de expresiones

```
1 func (s *Sym) Eval(envs *StkEnv) (val *Sym) {
2     if s.sType != SVar {
3         val, _ = NewGSym("", SConst)
4     }
5     switch s.sType {
6     case SConst:
7         envs.dprintf("SConst\n")
8         val.CopyValFrom(s)
9     case SVar:
10        envs.dprintf("SVar\n")
11        v := envs.GetSym(s.name)
12        val = v.val
```

Ejemplo calculadora

- Evaluación de expresiones

```
13  case SBinary :
14      envs.dprintf("SBinary\n")
15      left := s.Expr.Left.Eval(envs)
16      val.CopyValFrom(left)
17      right := s.Expr.Right.Eval(envs)
18      val.BinExpr(right, s.Expr.Op)
19  case SUnary :
20      envs.dprintf("SUUnary\n")
21      left := s.Expr.Left.Eval(envs)
22      val.CopyValFrom(left)
23      val.UnaryExpr(s.Expr.Op)
24  default :
25      panic("not a value")
26  }
27  envs.dprintf("\t——> %s val: %v\n", s, val)
28  return val
29 }
```