Parseo y Generación de Código

17 de agosto de 2017

Introducción a la materia
Intérpretes y compiladores
Lenguajes formales

Licenciatura en Informática con Orientación en Desarrollo de Software
Universidad Nacional de Quilmes

Introducción a la materia

Introducción a la materia

Programa de la materia

1. Estructura de un compilador.
   Fases de un compilador, compilación vs. interpretación.
2. Análisis léxico.
   Lenguajes regulares, expresiones regulares, autómatas finitos.
3. Análisis sintáctico.
   Gramáticas independientes del contexto, árboles de sintaxis abstracta, parsers descendentes y ascendentes, generadores de parsers.
4. Intérpretes.
   Ejemplos de interpretación para lenguajes imperativos, funcionales, orientados a objetos.
5. Análisis semántico.
   Tablas de símbolos, inferencia de tipos, unificación.

Introducción a la materia

Programa de la materia

6. Generación de código intermedio.
   Máquinas de pila, máquinas de registros, análisis y síntesis de atributos.
7. Análisis estático.
   Grafos de flujo, análisis de flujo de datos local hacia adelante y hacia atrás.
8. Optimización.
   Dead code elimination, *constant folding*, *loop unrolling*.
9. Soporte en tiempo de ejecución.
   Representaciones *tagged* vs. *tagless*, registros de activación, *garbage collection*.
10. Generación de código.
    Register allocation, *spilling*, selección de instrucciones.

Introducción a la materia

Modalidad de evaluación

1. Dos parciales:
   - 1.1 *Primer parcial:* jueves 12 de octubre. (Lenguajes regulares, análisis sintáctico).
   - 1.2 *Segundo parcial:* jueves 30 de noviembre. (Intérpretes, tipado, generación de código, análisis estático).
   - 1.3 *Recuperatorio del primer parcial:* jueves 07 de diciembre.
   - 1.4 *Recuperatorio del segundo parcial:* jueves 14 de diciembre.
2. Dos TPs.
   - 2.1 *Primer TP:* Programar un generador de parsers.
   - 2.2 *Segundo TP:* Programar un generador de código para un lenguaje pequeño.

Introducción a la materia

Modalidad de evaluación

- Los parciales son a libro abierto.
- Los TPs se hacen individualmente o en pareja.
- Los parciales se califican con nota numérica.
- Los TPs se califican como aprobados o desaprobados.
- Los TPs desaprobados admiten una reentrega.
- Para aprobar la materia:
  - Ambos TPs aprobados.
  - ≥ 4 en ambos parciales.
- Para promocionar:
  - Ambos TPs aprobados.
  - ≥ 6 en ambos parciales.
  - Promedio ≥ 7 entre ambos parciales.

Introducción a la materia

Organización

- Lista de e-mails:
  `lds-est-parse@listas.unq.edu.ar`
- Sitio web de la materia:
  `https://sites.google.com/site/unqpgc/`

Introducción a la materia

Bibliografía

Básica:

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools* (2nd Edition). Addison Wesley, 2006.
- Andrew W. Appel. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

Complementaria:

- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.*
- Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.
- Christian Queinnec. *Lisp in Small Pieces.*
- Guy L. Steele. *RABBIT: A Compiler for SCHEME.*
- Adele Goldberg, David Rodson. *Smalltalk-80: The Language and Its Implementation.*

Intérpretes y compiladores

Intérpretes y compiladores

- De manera muy abstracta un **lenguaje de programación** es una correspondencia entre **programas** y sus **significados**.
- Por ejemplo, el lenguaje de programación C establece el significado del siguiente programa:

```
int main() {
    int a = 21;
    return a * 2;
}
```

- ¿Qué significa un programa? Es una pregunta compleja.

Intérpretes y compiladores

Hay varias maneras de decir cuál es el significado de un programa, dependiendo de la perspectiva, el paradigma de programación, el objetivo, etc. Por ejemplo:

- En un lenguaje de bajo nivel como C, podemos pensar que el significado de un programa es el conjunto de todos sus posibles efectos, dependiendo de la configuración inicial de la memoria.
- En un lenguaje orientado a objetos, podemos pensar que el significado de un programa es una descripción de potenciales interacciones entre objetos.
- En un lenguaje funcional puro como Haskell, podemos pensar que el significado de un programa es un valor perteneciente a algún tipo de datos.

En esta materia no vamos a hacerlo de manera matemáticamente precisa, pero es posible hacerlo.

Intérpretes y compiladores

- Si P es un programa escrito en el lenguaje $\mathcal{L}$ y X es una lista de parámetros, vamos a escribir $\mathcal{L}(P, X)$ para denotar el resultado que tiene la ejecución de P con esos parámetros de acuerdo con la especificación del lenguaje $\mathcal{L}$.
- La terminología ("parámetros", "resultado") está usada de manera general. Por ejemplo, en un lenguaje de bajo nivel, los parámetros corresponden a la configuración de la memoria, y el resultado es una serie de efectos.

Intérpretes y compiladores

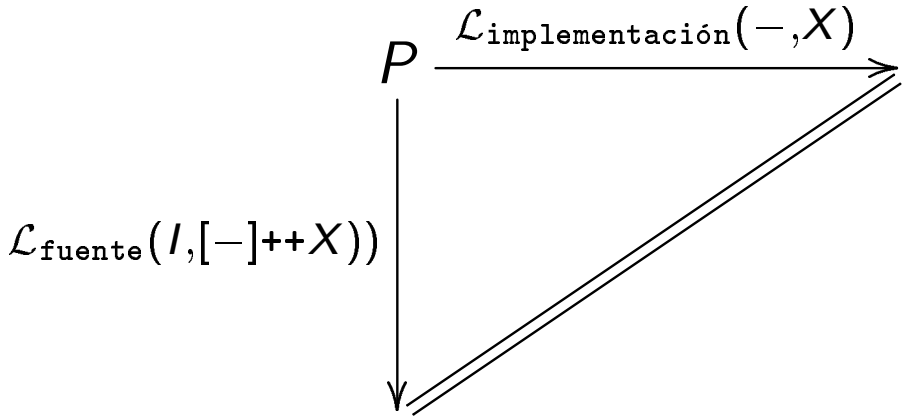
Intérpretes

Un *intérprete* para un lenguaje $\mathcal{L}_{\texttt{fuente}}$ es un programa I escrito en un lenguaje $\mathcal{L}_{\texttt{implementación}}$, que cumple la siguiente igualdad:

$$\mathcal{L}_{\texttt{fuente}}(P, X) = \mathcal{L}_{\texttt{implementación}}(I, [P]\texttt{++}X)$$

para cualquier lista de parámetros X.

Es decir: el significado del programa P en el lenguaje $\mathcal{L}_{\texttt{fuente}}$ es el mismo que el significado del intérprete I en el lenguaje $\mathcal{L}_{\texttt{implementación}}$ cuando se le pasa el programa P como parámetro.



Intérpretes y compiladores

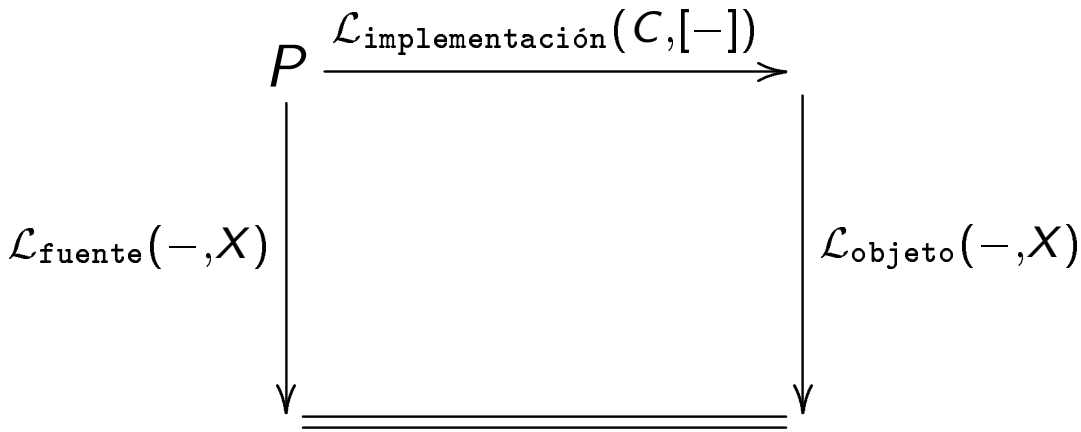
Compiladores

Un *compilador* para un lenguaje $\mathcal{L}_{\texttt{fuente}}$ es un programa C escrito en un lenguaje $\mathcal{L}_{\texttt{implementación}}$ que genera como resultado un programa escrito en un lenguaje $\mathcal{L}_{\texttt{objeto}}$ y cumple la siguiente igualdad:

$$\mathcal{L}_{\texttt{fuente}}(P, X) = \mathcal{L}_{\texttt{objeto}}(\mathcal{L}_{\texttt{implementación}}(C, [P]), X)$$

para cualquier lista de parámetros X.

El significado del programa P en el lenguaje $\mathcal{L}_{\texttt{fuente}}$ es el mismo que el significado del programa compilado en el lenguaje $\mathcal{L}_{\texttt{objeto}}$.



Intérpretes y compiladores

- Las definiciones que dimos arriba sólo tienen finalidad ilustrativa.
- No vamos a *usarlas*.
- Ser un intérprete o un compilador no es una propiedad de un lenguaje, sino de sus implementaciones.
- La realidad es compleja: muchos lenguajes modernos usan técnicas híbridas de interpretación y compilación (p.ej. compilación a *bytecode*, compilación *just-in-time*).
- Pero es importante entender esta distinción tradicional.

El lenguaje Calc-A

Vamos a trabajar con una calculadora de juguete, Calc-A.
Un programa en Calc-A es una lista de bits.
Cuando el programa se ejecuta, dispone de una pila de números enteros en la que se pueden meter o sacar números, y operar con ellos.
Las instrucciones se codifican así:

- 0: meter un número en la pila.
  Los tres dígitos siguientes indican el número codificado en binario (000 mete un 0, ..., 111 mete un 7).
- 1: ejecutar una operación.
  Los dos dígitos siguientes indican qué operación.
  - 00 sumar los dos números en el tope de la pila y meter el resultado.
  - 01 multiplicar los dos números en el tope de la pila y meter el resultado.
  - 10 sacar el número del tope de la pila y meter su opuesto.
  - 11 operación sin efecto.

El lenguaje Calc-A

Ejemplo de programa en Calc-A

Ejecutar el programa 011101110001110100101.

Intérprete de Calc-A en Python

Podemos escribir el siguiente intérprete de Calc-A en Python:

```
def interpretar(p):
    s = []
    while p != '':
        i = p[0:4] if p[0] == '0' else p[0:3]
        if i == '111': pass
        elif i == '0000': s.append(0)
        elif i == '0001': s.append(1)
        elif i == '0010': s.append(2)
        elif i == '0011': s.append(3)
        elif i == '0100': s.append(4)
        elif i == '0101': s.append(5)
        elif i == '0110': s.append(6)
        elif i == '0111': s.append(7)
        elif i == '100': s.append(s.pop() + s.pop())
        elif i == '101': s.append(s.pop() * s.pop())
        elif i == '110': s.append(-s.pop())
        p = p[len(i):]
    return s
```

Intérprete de Calc-A en Haskell

O el siguiente intérprete de Calc-A en Haskell:

```
type ProgramaCalcA = [Char]
type Pila = [Integer]

interpretar :: ProgramaCalcA -> Pila
interpretar p = i p []
  where
    i "" s                                  = s
    i ('0' : '0' : '0' : '0' : p) s         = i p (0 : s)
    i ('0' : '0' : '0' : '1' : p) s         = i p (1 : s)
    i ('0' : '0' : '1' : '0' : p) s         = i p (2 : s)
    i ('0' : '0' : '1' : '1' : p) s         = i p (3 : s)
    i ('0' : '1' : '0' : '0' : p) s         = i p (4 : s)
    i ('0' : '1' : '0' : '1' : p) s         = i p (5 : s)
    i ('0' : '1' : '1' : '0' : p) s         = i p (6 : s)
    i ('0' : '1' : '1' : '1' : p) s         = i p (7 : s)
    i ('1' : '0' : '0' : p) (a : b : s) = i p (a + b : s)
    i ('1' : '0' : '1' : p) (a : b : s) = i p (a * b : s)
    i ('1' : '1' : '0' : p) (a : s)         = i p (-a : s)
    i ('1' : '1' : '1' : p) s               = i p s
```

Compilador de Calc-A en Haskell a Python

Por otro lado, podemos hacer un compilador de Calc-A a Python:

```
type ProgramaCalcA    = [Char]
type ProgramaPython = String

compilarAPython :: ProgramaCalcA -> ProgramaPython
compilarAPython p = "s = []\n" ++ c p ++ "print(s)\n"
 where
  c ""                    = ""
  c ('0':'0':'0':'0':p) = "s.append(0)\n" ++ c p
  c ('0':'0':'0':'1':p) = "s.append(1)\n" ++ c p
  c ('0':'0':'1':'0':p) = "s.append(2)\n" ++ c p
  c ('0':'0':'1':'1':p) = "s.append(3)\n" ++ c p
  c ('0':'1':'0':'0':p) = "s.append(4)\n" ++ c p
  c ('0':'1':'0':'1':p) = "s.append(5)\n" ++ c p
  c ('0':'1':'1':'0':p) = "s.append(6)\n" ++ c p
  c ('0':'1':'1':'1':p) = "s.append(7)\n" ++ c p
  c ('1':'0':'0':p) = "s.append(s.pop() + s.pop())\n" ++ c p
  c ('1':'0':'1':p) = "s.append(s.pop() * s.pop())\n" ++ c p
  c ('1':'1':'0':p) = "s.append(-s.pop())\n" ++ c p
  c ('1':'1':'1':p) = c p
```

El lenguaje Calc-B

A continuación vamos a trabajar con una calculadora (ligeramente) más realista, que llamamos Calc-B.

- Permite escribir números arbitrariamente largos y expresiones aritméticas usando como operadores binarios la suma $+$ y el producto $*$.
- Permite utilizar paréntesis para asociar.
- Permite poner espacios en blanco, que se ignoran.
- La precedencia de operadores se debe respetar; por ejemplo, $1 + 2 * 3 = 1 + (2 * 3)$.

Ejemplos de programas en este lenguaje:

$$0 + 9$$

$$11 + 22 * 33 + 44 * 55$$

$$11 + 22 * (33 + 44 * 55)$$

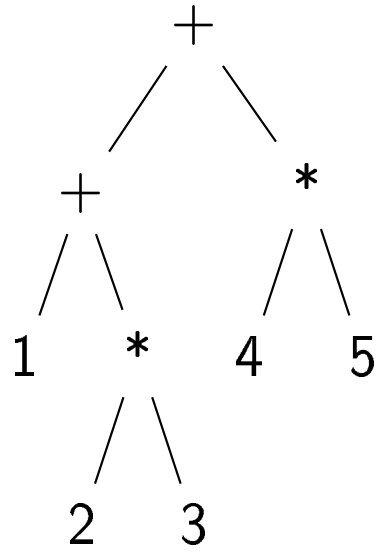
$$(((((42)))))$$

El lenguaje Calc-B

Queremos programar un compilador de Calc-B a Calc-A.
Este problema es bastante más complejo. Conviene dividirlo en dos grandes partes:

- Análisis sintáctico del programa en Calc-B para obtener un árbol. ("Parseo").
- Procesamiento del árbol obtenido para generar código en Calc-A. ("Generación de código").

Esquema del compilador

Entrada	$1 + 2 * 3 + 4 * 5$
Parseo	
Árbol de sintaxis	
Generación de código	
Salida	00010010011110110001000101101100

El analizador sintáctico

Para programar un analizador sintáctico tenemos que decidir cómo representar los árboles sintácticos ("ASTs").
Para programarlo en Haskell podríamos elegir la siguiente representación:

```
data AST =
       Constante Integer
     | Suma AST AST
     | Producto AST AST
```

El analizador sintáctico

Para programarlo en Python podríamos elegir la siguiente representación:

```
class Constante(object):
    def __init__(self, valor):
        self.valor = valor

class Suma(object):
    def __init__(self, ast1, ast2):
        self.ast1 = ast1
        self.ast2 = ast2

class Producto(object):
    def __init__(self, ast1, ast2):
        self.ast1 = ast1
        self.ast2 = ast2
```

El analizador sintáctico en Haskell

Usamos la técnica de **análisis sintáctico por descenso recursivo**.
Hay otras técnicas que vamos a ver más adelante.

```
parsear :: String -> AST
parsear p =
  let (ast, _) = parsearExpresion p
    in ast

comerEspacios :: String -> String
comerEspacios (' ' : s)  = comerEspacios s
comerEspacios ('\r' : s) = comerEspacios s
comerEspacios ('\n' : s) = comerEspacios s
comerEspacios ('\t' : s) = comerEspacios s
comerEspacios s          = s

esUnDigito :: Char -> Bool
esUnDigito c = '0' <= c && c <= '9'
```

El analizador sintáctico en Haskell

```
parsearExpresion :: String -> (AST, String)
parsearExpresion programa =
  let (termino, resto) = parsearTermino programa
      resto1 = comerEspacios resto
   in case resto1 of
        '+' : resto2 ->
          let (expresion, resto3) = parsearExpresion resto2
            in (Suma termino expresion, resto3)
        _ -> (termino, resto1)

parsearTermino :: String -> (AST, String)
parsearTermino programa =
  let (factor, resto) = parsearFactor programa
      resto1 = comerEspacios resto
   in case resto1 of
        '*' : resto2 ->
          let (termino, resto3) = parsearTermino resto2
            in (Producto factor termino, resto3)
        _ -> (factor, resto1)
```

El analizador sintáctico en Haskell

```
parsearFactor :: String -> (AST, String)
parsearFactor programa =
  let resto = comerEspacios programa in
    case resto of
      '(' : resto1 ->
        let (expresion, resto2) = parsearExpresion resto1
         in case resto2 of
              ')' : resto3 -> (expresion, resto3)
              _ -> error "Se esperaba un par ntesis."
      _ ->
        let (numero, resto2) = parsearNumero resto
         in (Constante (read numero :: Integer), resto2)

parsearNumero :: String -> (String, String)
parsearNumero (d : resto) =
  if esUnDigito d
   then let (ds, resto2) = parsearNumero resto
         in (d : ds, resto2)
   else ("", d : resto)
parsearNumero "" = ("", "")
```

El analizador sintáctico en Python

Vamos a usar la misma técnica de análisis sintáctico.
Lo hacemos en Haskell y en Python porque comparar las dos versiones puede ser ilustrativo y aclarar el funcionamiento del parser.

```
def es_un_digito(c):
    return '0' <= c <= '9'

class Parser(object):
    def __init__(self, programa):
        self.s = programa
        self.i = 0

    def comer_espacios(self):
        while self.i < len(self.s) and \
              self.s[self.i] in [' ', '\t', '\r', '\n']:
            self.i += 1
```

El analizador sintáctico en Python

```
def parsear_expresion(self):
    expresion = self.parsear_termino()
    self.comer_espacios()
    while self.i < len(self.s) and self.s[self.i] == '+':
        self.i += 1
        expresion = Suma(expresion, self.parsear_termino())
        self.comer_espacios()
    return expresion

def parsear_termino(self):
    termino = self.parsear_factor()
    self.comer_espacios()
    while self.i < len(self.s) and self.s[self.i] == '*':
        self.i += 1
        termino = Producto(termino, self.parsear_factor())
        self.comer_espacios()
    return termino
```

El analizador sintáctico en Python

```
def parsear_factor(self):
    self.comer_espacios()
    if self.i < len(self.s) and self.s[self.i] == '(':
        self.i += 1
        factor = self.parsear_expresion()
        if self.i < len(self.s) and self.s[self.i] != ')':
            raise Exception("Se esperaba un parentesis.")
        self.i += 1
        return factor
    else:
        return Constante(int(self.parsear_numero()))

def parsear_numero(self):
    numero = ''
    while self.i < len(self.s) and es_un_digito(self.s[self
        numero += self.s[self.i]
        self.i += 1
    return numero
```

Un intérprete de Calc-B en Haskell

Una vez que convertimos la entrada en un AST, es inmediato programar un intérprete de Calc-B. Este paso no es necesario pero a veces puede ser útil.
El intérprete en Haskell:

```
evaluar :: AST -> Integer
evaluar (Constante n)  = n
evaluar (Suma a b)     = evaluar a + evaluar b
evaluar (Producto a b) = evaluar a * evaluar b
```

Un intérprete de Calc-B en Python

El intérprete en Python:

```
class Constante(object):
    ...
    def evaluar(self):
        return self.valor

class Suma(object):
    ...
    def evaluar(self):
        return self.ast1.evaluar() + self.ast2.evaluar()

class Producto(object):
    ...
    def evaluar(self):
        return self.ast1.evaluar() * self.ast2.evaluar()
```

Compilador de Calc-B a Calc-A en Haskell

Por último, podemos programar el compilador.

El compilador en Haskell:

```
compilar :: AST -> ProgramaCalcA
compilar (Constante n) = enBase7 n
  where
    enBase7 n | n < 7 = digito n
    enBase7 n = enBase7 (n `div` 7) ++
                "0111101" ++
                digito (n `mod` 7) ++
                "100"
    digito 0 = "0000"
    digito 1 = "0001"
    digito 2 = "0010"
    digito 3 = "0011"
    digito 4 = "0100"
    digito 5 = "0101"
    digito 6 = "0110"
compilar (Suma a b)     = compilar a ++ compilar b ++ "100"
compilar (Producto a b) = compilar a ++ compilar b ++ "101"
```

Compilador de Calc-B a Calc-A en Python

El compilador en Python:

```
def escribir_digito(n, salida):
    digitos = {
        0: '0000',
        1: '0001',
        2: '0010',
        3: '0011',
        4: '0100',
        5: '0101',
        6: '0110',
    }
    salida.write(digitos[n])

def escribir_en_base_7(n, salida):
    if n < 7:
        escribir_digito(n, salida)
    else:
        escribir_en_base_7(n // 7, salida)
        salida.write('0111101')
        escribir_digito(n % 7, salida)
        salida.write('100')
```

Compilador de Calc-B a Calc-A en Python

```
class Constante(object):
    ...
    def compilar(self, salida):
        escribir_en_base_7(self.valor, salida)

class Suma(object):
    ...
    def compilar(self, salida):
        self.ast1.compilar(salida)
        self.ast2.compilar(salida)
        salida.write('100')

class Producto(object):
    ...
    def compilar(self, salida):
        self.ast1.compilar(salida)
        self.ast2.compilar(salida)
        salida.write('101')
```

Estructura de un compilador

- *Preprocesamiento.*
  Fase que usan algunos lenguajes para expandir macros, incluir *headers*, detectar *pragmas*.
- *Análisis léxico.*
  Descompone la entrada en una lista de símbolos terminales o *tokens*, ignora espacios en blanco (si corresponde), elimina los comentarios.
- *Análisis sintáctico.*
  Construye un AST a partir de la lista de símbolos terminales.

Estructura de un compilador

- *Análisis semántico del AST.*
  Analiza el AST para asegurar que el programa está correctamente formado: p.ej. verifica de que todas las funciones que se usan estén declaradas, chequea o infiere los tipos, recolecta información sobre los símbolos para usarla en etapas posteriores del compilador.
- *Desugaring.*
  Los compiladores de algunos lenguajes funcionales tienen una etapa de "*desugaring*" en la que muchas construcciones complejas del lenguaje se reescriben en términos de unas pocas construcciones simples. P.ej. **let x = A in B** se puede reescribir como $(\lambda \mathbf{x.B})\ \mathbf{A}$.
- *Generación de código intermedio.*
  A partir del AST se genera una representación intermedia del programa. ¿Cuál es la ventaja de generar código intermedio en lugar de generar código máquina directamente?

Estructura de un compilador

- *Análisis del código intermedio.*
  Se puede hacer análisis estático del código intermedio para detectar patrones en el comportamiento del código, por ejemplo: variables declaradas que no se usan, fragmentos del código que nunca se ejecutan, expresiones que se calculan dos veces, valores que se calculan repetidamente dentro de un ciclo, etc.
- *Optimización.*
  El resultado de los análisis anteriores se puede aprovechar para hacer optimizaciones. Hay muchas optimizaciones posibles: propagación de constantes, desenrolle de ciclos, *inlining* de funciones.
- *Generación de código.*
  Finalmente se puede producir código a partir del código intermedio. Esto tiene sus propias complejidades, por ejemplo: reserva y *spilling* de registros, selección de instrucciones.

Comentarios

Las técnicas de parseo y generación de código se pueden usar para muchos fines, no sólo para diseñar lenguajes de propósito general. Por ejemplo:

- Lenguajes de consulta o búsqueda (p.ej. expresiones regulares).
- Lenguajes de *markup* (p.ej. XML y afines).
- Lenguajes específicos para distintos dominios (p.ej. SQL, LaTeX).
- Lenguajes de configuración de herramientas (p.ej. Snort).
- Herramientas de álgebra computacional, demostradores de teoremas.

Comentarios

Otro tema completamente distinto es el análisis sintáctico de lenguajes naturales.

No vamos a enfocarnos en eso en esta materia.

Lenguajes formales

Alfabetos y palabras

Si Σ es un conjunto finito de símbolos al que llamamos **alfabeto**, una **palabra** o **cadena** en el alfabeto Σ es una secuencia finita de símbolos $a_1 \ldots a_n$.

Escribimos ϵ para la palabra vacía, es decir cuando $n = 0$.

Si α es una palabra, notamos $|\alpha|$ a su longitud.

Si α, β son palabras, notamos $\alpha \cdot \beta$ o simplemente $\alpha\,\beta$ a su concatenación.

Si α es una palabra, notamos α^n al resultado de concatenar n veces consigo misma. Más precisamente:

$$\begin{aligned} \alpha^0 &= \epsilon \\ \alpha^{n+1} &= \alpha \cdot \alpha^n \end{aligned}$$

Si α es una palabra, notamos α^r a su reverso.

Notamos $\Sigma^\star$ para el conjunto de todas las palabras en el alfabeto Σ.

Alfabetos y palabras

Ejemplo.
Si $\Sigma = \{a, b\}$, algunas palabras son a, b, aaa, $abaab$, bab, ϵ, con $|a| = 1$, $|abaab| = 5$, $|\epsilon| = 0$. Además:

$$aaa \cdot \epsilon = aaa$$

$$\epsilon \cdot bab = bab$$

$$aaa \cdot bab = aaabab$$

$$(ab)^3 = ababab$$

$$(aaabab)^r = babaaa$$

$$((aaabab)^r)^r = (babaaa)^r = aaabab$$

Lenguajes formales

Un **lenguaje** en un alfabeto Σ es un conjunto de palabras $L \subseteq \Sigma^\star$.
Por ejemplo:

- Palabras en Σ de longitud menor que 10:

$$L_1 = \{\alpha \in \Sigma^\star : |\alpha| < 10\}$$

- Palabras capicúa:

$$L_2 = \{\alpha \in \Sigma^\star : \alpha = \alpha^r\}$$

- Palabras que tienen algún 0, con $\Sigma = \{0, 1\}$:

$$L_3 = \{\alpha \in \{0,1\}^\star : \text{existen } \beta, \gamma \in \{0,1\}^\star \text{ tales que } \alpha = \beta 0 \gamma\}$$

Gramáticas independientes del contexto

Nos interesa contar con herramientas para describir lenguajes. Por ejemplo, describir la gramática de un lenguaje de programación.

Una **gramática independiente del contexto** es una 4-upla (N, Σ, P, S), donde:

- N es un conjunto finito de símbolos, los **símbolos no terminales**.
- Σ es un conjunto finito de símbolos disjunto de N, los **símbolos terminales**.
- P es un conjunto de **producciones**. Cada producción es de la forma $A \to \beta$ donde $A \in N$ es un símbolo no terminal y $\beta \in (N \cup \Sigma)^\star$ es una cadena que potencialmente incluye símbolos terminales y no terminales.
- $S \in N$ es un símbolo no terminal, el **símbolo inicial**.

Gramáticas independientes del contexto

- Si (N, Σ, P, S) es una gramática definimos una relación binaria de **derivación en un paso** entre cadenas de símbolos:

$$\Rightarrow \subseteq (N \cup \Sigma)^\star \times (N \cup \Sigma)^\star$$

de la siguiente manera:

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

$$\text{si } (A \to \gamma) \in P \text{ es una producción}$$

- La relación de **derivación en muchos pasos** entre cadenas de símbolos:

$$\Rightarrow^\star \subseteq (N \cup \Sigma)^\star \times (N \cup \Sigma)^\star$$

se define como la clausura reflexiva–transitiva de $\Rightarrow$, es decir:
  - Vale $\alpha \Rightarrow^\star \alpha$ para cualquier cadena $\alpha \in (N \cup \Sigma)^\star$.
  - Si $\alpha \Rightarrow \beta$ y $\beta \Rightarrow^\star \gamma$, entonces $\alpha \Rightarrow^\star \gamma$.
- Una derivación se dice **más a la izquierda** si en cada paso se elige una producción asociada al no terminal que aparece más a la izquierda de la cadena.

Gramáticas independientes del contexto

Por ejemplo, si:

- El conjunto de símbolos terminales es $\Sigma = \{\mathbf{n}, +, *, (,)\}$.
- El conjunto de símbolos no terminales es $N = \{E, T, F\}$.
- El símbolo inicial es E.
- El conjunto de producciones P es el siguiente:

$$\begin{array}{lcl} E & \to & E + T \\ E & \to & T \\ T & \to & T * F \\ T & \to & F \\ F & \to & (E) \\ F & \to & \mathbf{n} \end{array}$$

Dar una derivación más a la izquierda de $E \Rightarrow^\star \mathbf{n} + (\mathbf{n} * \mathbf{n})$.

Gramáticas independientes del contexto

- Si $S \Rightarrow^\star \alpha$ donde S es el símbolo inicial de una gramática G, decimos que α es una **forma sentencial** de G.
- Observar que una forma sentencial α puede incluir símbolos terminales y no terminales.
- Una **sentencia** de G es una forma sentencial que no incluye símbolos no terminales.
- El **lenguaje generado por una gramática** G es el conjunto de sentencias de G. Notamos $L(G)$ al lenguaje generado por G.
- Dicho de otro modo, una cadena de símbolos terminales $\alpha \in \Sigma^\star$ está en $L(G)$ si y sólo si $S \Rightarrow^\star \alpha$.

Gramáticas

Ejercicios.

- Dar una gramática independiente del contexto que genere el lenguaje L_1 de las palabras de longitud menor que 10.
- Dar una gramática independiente del contexto que genere el lenguaje L_2 de las palabras capicúa.
- Dar una gramática independiente del contexto que genere el lenguaje L_3 de las palabras que tienen algún 0.