Lenguajes y Compiladores
Introducción

Miguel Pagano

12 de marzo de 2025

¿Qué hace un compilador?

De un texto a una estructura de datos

Un programa en Python

```
def factorial(n):
  if n <= 1:
    return   1
  else:
   return factorial(n-1) * n
```

¿Qué ve el intérprete cuando usamos esa función?

```
>>> texto = """def factorial(n):
...     if n <= 1:
...     …
... """
>>> funcion = ast.parse(texto).body[0]
```

El output

```
>>> print(ast.dump(funcion.body[0],indent=2))
If(
  test=Compare(
    left=Name(id='n', ctx=Load()),
    ops=[
      LtE()],
    comparators=[
      Constant(value=1)]),
  body=[
    Return(
      value=Constant(value=1))],
  orelse=[
    Return(
      value=BinOp(
        left=Call(
          func=Name(id='factorial', ctx=Load()),
          args=[
            BinOp(
              left=Name(id='n', ctx=Load()),
              op=Sub(),
              right=Constant(value=1))],
          keywords=[]),
        op=Mult(),
        right=Name(id='n', ctx=Load()))))])
```

Fases de un compilador

Lexer
Reconoce los elementos sintácticos, ***tokens***, válidos:

```
>>> g = tokenize.tokenize(BytesIO(texto.encode('utf-8')).readline)
>>> for i in g:
...     print(i)
TokenInfo(type=1 (NAME), string='def', start=(1, 0), end=(1, 3), line='
TokenInfo(type=1 (NAME), string='factorial', start=(1, 4), end=(1, 13),
TokenInfo(type=54 (OP), string='(', start=(1, 13), end=(1, 14), line='d
```

Los tokens válidos se describen con expresiones regulares.

Fases de un compilador

El parser

```
function_def:
 'def' NAME &&'(' [params] ')' && ':' block
```

A medida que se consume el input se genera un ***árbol de sintaxis abstracta***.

Fases de un compilador

La gramática abstracta
se concentra en la información relevante:

⟨*stmt*⟩ ::= ‘`FunctionDef`’ ⟨*identifier*⟩ ⟨*arguments*⟩ ⟨*stmt*⟩* ⟨*expr*⟩* ⟨*expr*⟩?
⟨*string*⟩?
| …

Sólo tenemos un ***constructor*** ‘`FunctionDef`’, un no-terminal ⟨*identifier*⟩ para el nombre, los argumentos (con su propia estructura), el cuerpo (que es una lista de ‘`stmt`’) y un par de argumentos más (opcionales).

Fases de un compilador

Análisis estático
Una vez que tenemos el árbol de sintaxis abstracta (AST) podemos analizarlo para detectar errores antes de ejecutar (por ejemplo, chequeo de tipos) o para optimizarlo (por ejemplo, ¿hay código inútil?).

Generación de código ó Interpretación
Finalmente, se genera código de bajo nivel a partir de un AST.

Reconectando con la materia

¿Cómo le damos sentido a un lenguaje?

1. Conociendo la ***sintaxis abstracta*** del lenguaje.
2. Explicando el ***sentido*** de cada construcción del lenguaje.

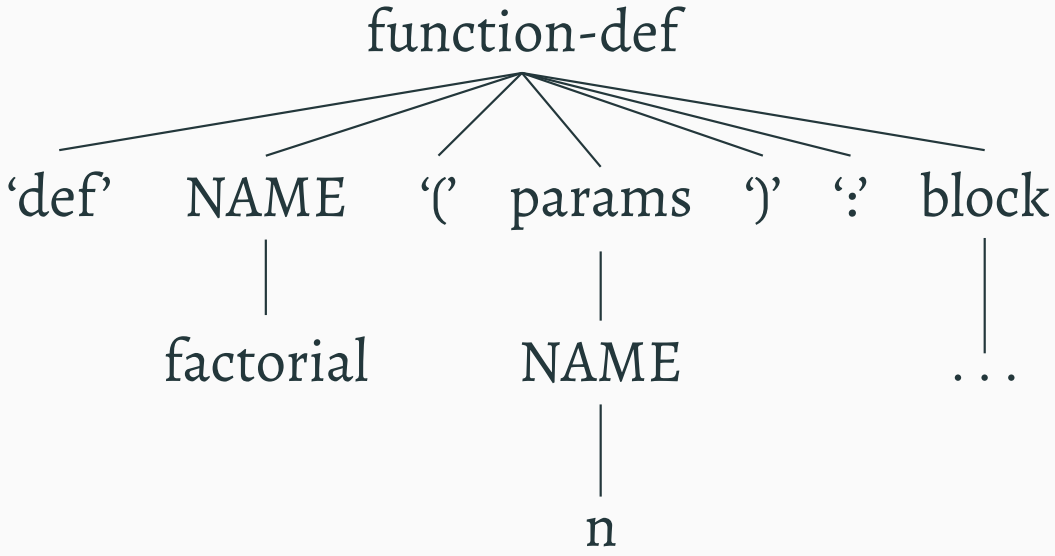
Descripciones de un lenguaje

1. ***Informal***: "The unary - (minus) operator yields the negation of its numeric argument; the operation can be overridden with the `__neg__()` special method.".
2. ***Axiomática***: Si $P \Rightarrow Q[x := E]$, entonces $\{P\}\ x\ :=\ E\ \{Q\}$.
3. ***Operacional***: Si $e \rightsquigarrow \lfloor k \rfloor$, entonces $-e \rightsquigarrow \lfloor -k \rfloor$.
4. ***Denotacional***: lo que veremos a continuación.

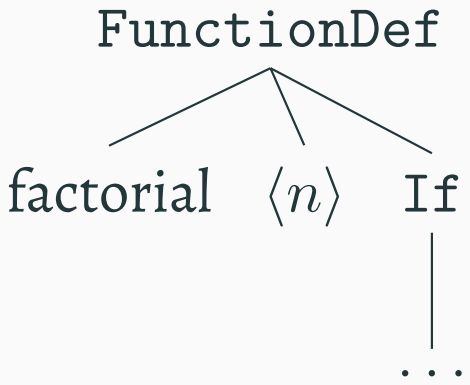
Sintaxis Abstracta

Dos gramáticas, dos árboles

Concreta: demasiada información irrelevante



Abstracta: sólo lo necesario para analizar la frase



¿Por qué Sintaxis Abstracta?

Queremos una notación matemática concisa que nos permita dar sentido a las construcciones del lenguaje sin preocuparnos de cómo se escriben.

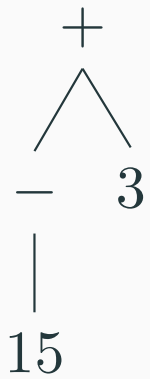
¿Por qué Sintaxis Abstracta?

$\langle exp \rangle$ ::= 0 | 1 | 2 | ...
| - $\langle exp \rangle$
| $\langle exp \rangle$ * $\langle exp \rangle$
| $\langle exp \rangle$ + $\langle exp \rangle$

Consideremos la expresión: $-15 + 3$.

¿Por qué Sintaxis Abstracta?

1. Diremos que $\langle exp \rangle$ es una ***categoría sintáctica*** de nuestro lenguaje.
2. Si bien lo escribimos linealmente, pensamos en árboles.
3. Como lo escribimos linealmente, puede haber ambigüedad.
4. Resolvemos la ambigüedad mediante reglas de asociatividad, precedencia, y usando paréntesis (sólo en caso de mucha necesidad).



Los árboles no son árboles de derivación: sólo ponemos los ***constructores*** (terminales) y los sub-árboles de sus ***argumentos***.

¿Por qué Sintaxis Abstracta?

Nada nuevo bajo el sol
Esto es lo que venimos haciendo desde primer año, con un poco de problematización en IntroLog (y volviendo para atrás en Lógica).

Sintaxis abstracta

Ahora más matemáticamente
En IntroLog teníamos fórmulas, ***la sintaxis*** que la presentábamos como un conjunto definido inductivamente).

Si volvemos a nuestro ejemplo $\langle exp \rangle$, podemos pensar que la gramática abstracta establece que un conjunto E es ***un*** conjunto de ***frases abstractas*** si al tomar un elemento $e \in E$

1. podemos reconocer que e es 0 ó 1 ó 2, etc, ó
2. podemos reconocer que es $-e'$, con $e' \in E$,
3. podemos reconocer a $e_1 + e_2$, con $e_1, e_2 \in E$.

Es decir, 0, 1,...son constantes en E y tanto $-\_ \colon E \to E$ como $\_ + \_ \colon E \times E \to E$ son funciones inyectivas.

Ejercicio: dar ejemplos concretos de conjuntos de frases abstractas.

Semántica

Sentido

Repaso
En IntroLog hacíamos una distinción muy tajante entre la "sintaxis" y la "semántica"; la sintaxis eran las fórmulas (pura forma) a las que les dábamos ***sentido*** en el conjunto $\{0, 1\}$ mediante la función $[\![\_]\!] : \mathit{Prop} \to \{0, 1\}$.

Al dar semántica, fórmulas disintas como $\top$ y $\bot \Rightarrow \bot$ quedaban igualadas.

¿Cómo definíamos $[\![\_]\!]$ para asegurarnos que estaba bien definida?

Sentido

Ecuaciones dirigidas por la sintaxis
Sin saberlo lo que hacíamos era dar una definición ***dirigida por la sintaxis***:

1. Para cada constructor había una ecuación y
2. el lado derecho estaba dado en función de la semántica de sus sub-frases inmediatas.

$$
\begin{array}{lcl}
[\![\_]\!] : \langle exp \rangle \to \mathbb{Z} & & \\
[\![0]\!] & = & 0 \\
[\![1]\!] & = & 1 \\
 & \vdots & \\
[\![-e]\!] & = & -[\![e]\!] \\
[\![e + e']\!] & = & [\![e]\!] + [\![e']\!] \\
[\![e * e']\!] & = & [\![e]\!] * [\![e']\!]
\end{array}
$$

Observaciones

1. En $[\![0]\!] = 0$ las dos ocurrencias de 0 tienen tipos distintos; a esto le llamamos ***meta-circularidad***: definimos el sentido de un símbolo por una entidad matemática (el cero) a la cual escribimos con ese mismo símbolo.
2. En $[\![-e]\!] = -[\![e]\!]$ estamos usando la meta-variable e, algo que pertenece al ***meta-lenguaje***, para referirnos al argumento de $-$.
3. Decimos que $\langle exp \rangle$ es el ***lenguaje objeto*** y usamos toda la matemática como meta-lenguaje, es decir como lenguaje para describirlo y estudiarlo.
4. La dirección por sintaxis asegura la ***composicionalidad*** de la semántica; es decir, si en una frase e cambio una subfrase e_1 por otra e_2 obteniendo e': si $[\![e_1]\!] = [\![e_2]\!]$, entonces $[\![e]\!] = [\![e']\!]$.

Un poco más abstracto

En lenguaje de Lógica
Damos una interpretación para la signatura de nuestro lenguaje de términos: elegimos un conjunto A e interpretamos cada terminal, es un símbolo de función, como una función de acuerdo a la aridad del símbolo.

En lenguaje de Haskell
Podemos pensar que la gramática abstracta está dada por un tipo de datos (ejercicio) y que la semántica no es otra cosa que el "fold" asociado a ese tipo.

Sintaxis como semántica
La interpretación de cada constructor como sí mismo nos da un modelo. De hecho, hay una única función desde el modelo sintáctico a cualquier otro modelo.

La necesidad de un estado

Consideremos este nuevo lenguaje:

$$
\begin{aligned}
\langle intexp \rangle ::= \; & \langle var \rangle \mid 0 \mid 1 \mid \ldots \\
\mid \; & -\langle intexp \rangle \\
\mid \; & \langle intexp \rangle * \langle intexp \rangle \mid \langle intexp \rangle \; / \; \langle intexp \rangle \mid \langle intexp \rangle \; \% \; \langle intexp \rangle \\
\mid \; & \langle intexp \rangle + \langle intexp \rangle \mid \langle intexp \rangle - \langle intexp \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle assert \rangle ::= \; & \textbf{true} \mid \textbf{false} \\
\mid \; & \langle intexp \rangle \text{ `=' } \langle intexp \rangle \mid \langle intexp \rangle < \langle intexp \rangle \mid \langle intexp \rangle \leqslant \langle intexp \rangle \\
\mid \; & \langle intexp \rangle \neq \langle intexp \rangle \mid \langle intexp \rangle > \langle intexp \rangle \mid \langle intexp \rangle \geqslant \langle intexp \rangle \\
\mid \; & \neg \langle assert \rangle \\
\mid \; & \langle assert \rangle \vee \langle assert \rangle \mid \langle assert \rangle \wedge \langle assert \rangle \\
\mid \; & \langle assert \rangle \Rightarrow \langle assert \rangle \mid \langle assert \rangle \equiv \langle assert \rangle \\
\mid \; & \exists \langle var \rangle \; . \; \langle assert \rangle \mid \forall \langle var \rangle \; . \; \langle assert \rangle
\end{aligned}
$$

$\langle var \rangle$ es un no-terminal sin constructores: es un conjunto numerable de “variables”.

La necesidad de un estado

Preguntas

- ¿Qué categorías sintácticas tiene el lenguaje?
- ¿Cuál es el dominio semántico “natural”?
- ¿Cuál es la diferencia con los símbolos de función de Lógica?
- ¿Las ecuaciones semánticas?