

SquanchyPL - Algoritmos y Computación

Conde Osorio, Marcos V.

1. Introducción

SquanchyPL es un lenguaje de programación de alto nivel desarrollado por Marcos Conde Osorio y Gabriel Rodríguez Canal, estudiantes de la Universidad de Valladolid. Este proyecto nace de la búsqueda de la comprensión del proceso de desarrollo de un lenguaje de programación y de la algoritmia que lo soporta. El lenguaje se crea desde 0, tanto su gramática basada en Python y Haskell como su compilador, que es el punto principal de este trabajo. Se ha trabajado sobre la publicación de Vaughan Pratt acerca de la precedencia de operadores publicada en los años 70, la documentación oficial de Python y de LLVM. El objetivo de Squanchy es ser un lenguaje de programación sencillo, claro, compilado y eficiente, que pueda usarse en el ámbito del análisis de datos como R o Python, de ahí su estilo imperativo-funcional.

Puedes consultar el estado actual del proyecto en el github del mismo y encontrar ejemplos de código, tutoriales ...

2. Breve explicación sobre gramáticas

Todo lenguaje de programación tiene una gramática que define su sintaxis. Las gramáticas están compuestas por símbolos y reglas de producción. La gramática de SquanchyPL es de tipo LL(1) (perteneciente a las CFG, más concretamente a las DCFG). Esto implica que con tan solo leer un símbolo del programa de entrada es posible identificar la regla de producción que le corresponde de forma unívoca (no sería una gramática LL(1), por tanto, aquella en la que existan ambigüedades).

Es conveniente aclarar antes de los ejemplos que la primera regla de producción que se aplica es siempre S (en la descripción formal de gramáticas).

El resto del proceso de *parsing* consiste en resolver los átomos no terminales con las reglas de producción.

A continuación se muestra un ejemplo de gramática LL(1) y de otra que no lo es (a, b, c y d son terminales):

$$W_1 = aaabd$$

En la figura 1 se muestra un ejemplo de gramática LL(1), esto es, es posible generar el *parser tree* leyendo los tokens de W_1 de uno en uno, siguiendo las reglas de producción sin ambigüedad. El resultado es el *parser tree* de la figura 2.

$$\begin{aligned} S &\rightarrow aA|bB \\ A &\rightarrow aB|cB \\ B &\rightarrow bC|aC \\ C &\rightarrow bD \\ D &\rightarrow d \end{aligned}$$

Figura 1: Gramática G_1

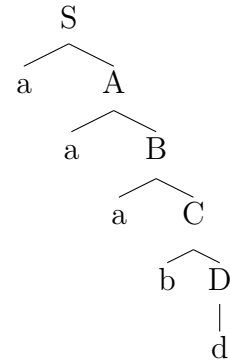


Figura 2: Parsing con G_1

$$W_2 = abd$$

En la figura 3, en cambio, se muestra una gramática que no es LL(1), pues la primera regla de producción, S, ofrece dos posibilidades, $abB|aaA$, que comienzan ambas con el mismo token. Se da, por tanto, una ambigüedad que no es posible solucionar con la lectura de un solo token. Si se pasa a considerar la gramática como LL(2) (se leen los tokens de dos en dos) sí es posible realizar el proceso de *parsing*, obteniéndose el *parser tree* de la figura 4.

$$\begin{aligned} S &\rightarrow abB|aaA \\ B &\rightarrow d \\ A &\rightarrow c|d \end{aligned}$$

Figura 3: Gramática G_2

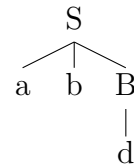


Figura 4: Parsing con G_2

3. Diseño del compilador

En esta sección se resumen las partes en que se divide un compilador con objeto de facilitar la comprensión del resto de la memoria. Todas ellas son modulares, es decir, independientes. Cada módulo recibe como entrada la salida del anterior. Así, el lexer recibe como entrada el código fuente del programa y devuelve la lista de tokens. El parser recibe como entrada la lista de tokens y devuelve el parse tree, y así sucesivamente.

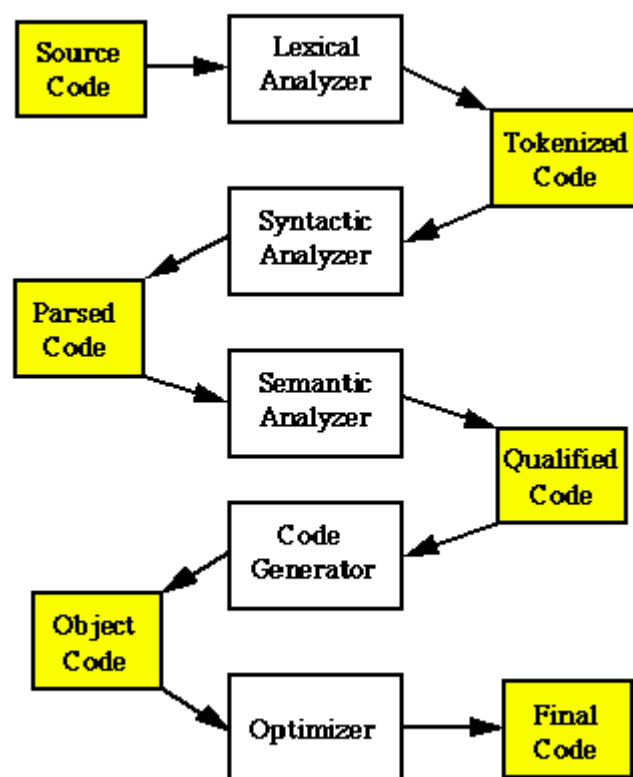


Figura 5: Wikipedia. Etapas de compilación

3.1. Analizador léxico. Lexer

En la sección 2 se introdujo el concepto de símbolo. En términos prácticos se habla de *tokens*. Mientras que el símbolo es un elemento de la gramática el *token* se podría describir como la instancia de ese símbolo en el programa. El lexer se encarga de obtener los *tokens* del programa de entrada, leyendo de

izquierda a derecha. Además, todos los espacios en blanco, líneas en blanco, comentarios y demás información innecesaria se elimina del programa fuente. Las expresiones regulares tienen especial interés para la implementación de *lexers*, enfoque utilizado en el lexer de SquanchyPL.

Por ejemplo:

```
programa = a + b * 2
tokens = [a, +, b, *, 2]
```

La salida real que proporciona nuestro lexer sería la siguiente:

```
program = suma (a,b) -> c :: c:a+b

tokens = [(Module, Module), (Name, suma), (operator, '('),
          (Name, a), (operator, ','), (Name, b), (operator, ')'),
          (operator, '$->$'), (Name, c), (operator, ':'), (Name, c),
          (operator, ':'), (Name, a), (operator, '+'), (Name, b),
          ((end), (end))]
```

Los tokens *Module* y *end* que indican el comienzo y final del código y no son escritos explícitamente por el usuario.

Destacar que en esta parte también se comprueba que los símbolos de la gramática están bien escritos, y se tratan los errores más sencillos. Por ejemplo:

```
in [1]: a $* b
out[1]: Unknown operator "$*"

in [2]: "hello world
out[2]: String Error. Missing '''
```

3.2. Analizador sintáctico. Parser

El parser es el módulo encargado de generar el *AST* a partir de la lista de tokens. El *AST* o *Abstract Syntax Tree*, como su propio nombre indica es un árbol que describe las relaciones entre los tokens delimitadas por la gramática del lenguaje.

La representación típica de un *AST* es un *string*, e.g.:

Expresión = $a + b * 3$
AST = Add (Name a , Mult (Name b , Const 3))

Nuestra representación del *AST* se realiza con la misma notación del módulo *compiler* de Python 2, y es perfectamente comparable al módulo *ast* de Python3, basta con ejecutar el siguiente código en Python:

```
import compiler
expression = "a+b*3"
compiler.parse(expression,"eval")
```

3.3. Analizador semántico

Una vez disponemos de toda la información relevante del código fuente ordenada jerárquicamente en forma de un *AST*, comprobamos restricciones de tipos y otras limitaciones semánticas antes de la generación de código. En nuestro caso este análisis se realiza en el *Parser*.

A continuación algunos ejemplos:

- En los lenguajes de tipado estático, puede darse un caso como el siguiente en el que el código está correctamente escrito desde el punto de vista léxico y sintáctico, pero da lugar a un error de tipo.

```
int a = "hello world"
```

En nuestro caso el tipado es dinámico, se infiere en la operación de asignación.

- Una posible limitación semántica se puede dar en la asignación, algunos lenguajes como JavaScript o Python permiten almacenar funciones en variables, en nuestro caso lo permitimos también pero se podría delimitar a que las variables solo pueden almacenar tipos primitivos.

3.4. Generación de código intermedio

EL código fuente puede ser traducido en su código de la máquina destino, entonces, ¿por qué hemos de traducir el código fuente en un código intermedio que luego se traduce en su código de destino? Vamos a ver las razones por las que necesitamos un código intermedio.

- Es una representación intermedia explícita del programa fuente, entre el alto nivel y ensamblador, destinado a una máquina abstracta, esto permite sencillez y transportabilidad dado que se puede 'crear' un compilador para cada máquina. En nuestro caso, lo interesante es poder utilizar el mismo compilador en diferentes arquitecturas: x86, arm, mips, PowerPC ...
- El código intermedio IR es fácil de producir y fácil de traducir a código objeto.
- IR es muy similar al ensamblador por lo que se puede leer y entender fácilmente.

¿Cómo generamos código intermedio? Utilizamos **LLVM** [24], una infraestructura para desarrollar compiladores, que suministra las capas intermedias en un sistema de compilado completo; se podría decir que es un toolkit para el desarrollo de compiladores. LLVM es muy conocida y utilizada por muchos lenguajes de programación como: Python, R, Haskell, Lisp, Ada, Fortran, C#, Squanchy ...

A continuación se puede apreciar un código muy simple en C, y su correspondiente código intermedio LLVM-IR.

```
int mul_add(int x, int y, int z) {
    return x * y + z;
}

define i32 @mul_add(i32 %x, i32 %y, i32 %z) {
entry:
    %tmp = mul i32 %x, %y
    %tmp2 = add i32 %tmp, %z
    ret i32 %tmp2
}
```

3.5. Optimización de código y Target Code

Tanto la optimización de código como el paso de código intermedio IR a código objeto se realiza también con LLVM, y de esta forma conseguimos desarrollar un compilador íntegro para Squanchy, que se puede utilizar además en distintas arquitecturas.

4. Implementación de SquanchyPL

Este trabajo está basado en el paper de Vaughan Pratt [19]. Se ha escogido Python3 para la implementación, dada la potencia de sus módulos de tratamiento de *strings* y su orientación al objeto. A continuación se describen las características más destacadas de la algoritmia de la solución implementada.

4.1. Lexer

4.1.1. Implementación

El lexer de SquanchyPL ha sido implementado utilizando expresiones regulares (librería **re**). Es un módulo completamente funcional desarrollado desde 0 y comparable con el módulo *Tokenize* de Python 2.

4.1.2. Análisis de coste

El análisis de coste del lexer no es directo, puesto que se han utilizado diferentes expresiones regulares y no se conoce la complejidad de las mismas. Se ha optado, por tanto, por un enfoque de estimación. El procedimiento seguido ha sido el siguiente:

1. Generar una cadena de n símbolos.
2. Pasar como argumento la cadena al parser.
3. Tomar tiempo de inicio.
4. Tomar tiempo de finalización.
5. Guardar la diferencia de tiempos (tiempo de finalización - tiempo de inicio)
6. Volver a 1 hasta iterar k veces.
7. Calcular la media de las diferencias de tiempos.

De este modo se obtiene el tiempo medio que el lexer tarda en procesar una entrada de tamaño n (donde el tamaño se mide en número de símbolos). En el cálculo se escogió siempre $k = 1000$.

La gramática de Python es también LL(1) y su lexer es de libre acceso, de modo que se puede efectuar una comparación entre el lexer de Python y el de SquanchyPL. Para ello el algoritmo de estimación de coste se repitió para diferentes valores de n , resultando gráfico 6:

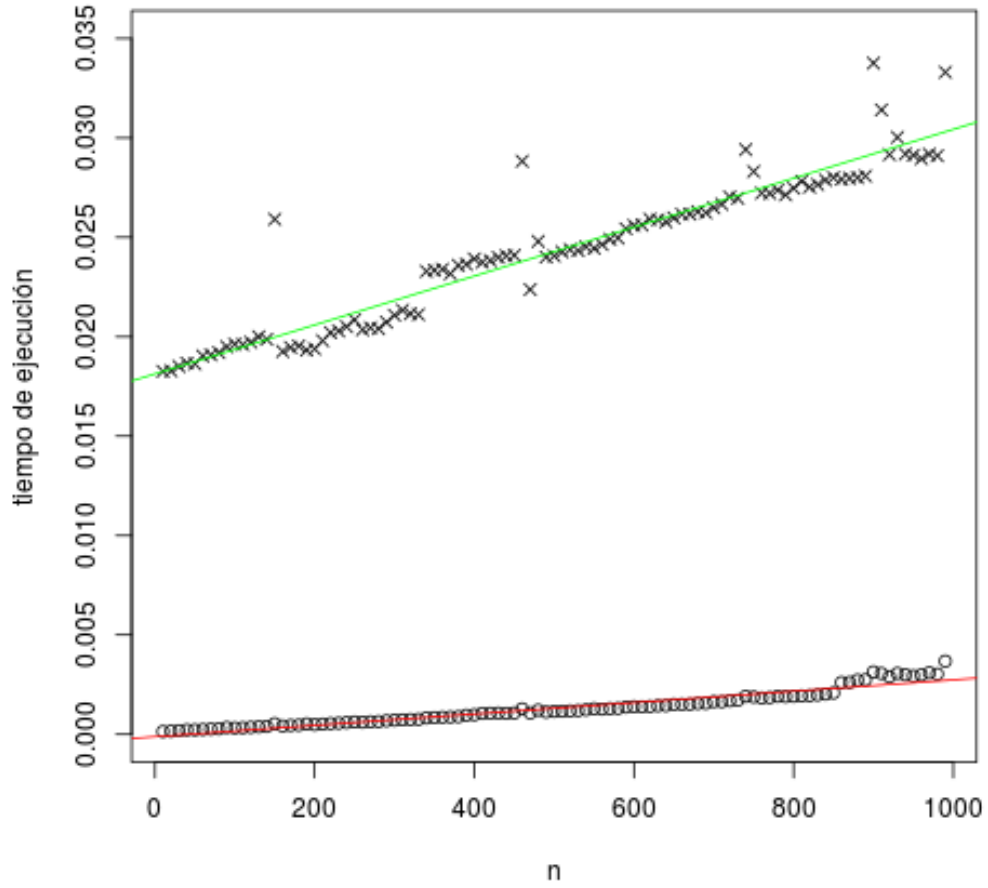


Figura 6: Tiempo de ejecución del lexer de Squanchy-PL (rojo) y el de Python (verde) vs n (longitud de la cadena)

Se aprecia que el tiempo de ejecución del lexer depende linealmente del tamaño de la entrada (él código), por lo que se puede afirmar que el coste del lexer es $\mathcal{O}(n)$. También se puede ver que el lexer de Squanchy es unas 10 veces más rápido que el lexer de Python, esto se debe a que la clasificación que realiza Python es mucho más exhaustiva que la nuestra.

4.2. El parser de Pratt

4.2.1. Implementación

El parser de Pratt [19] fue planteado para gramáticas LL(1), como es el caso del lenguaje de este trabajo, y concebido para lenguajes funciones. Sin embargo, nosotros lo hemos ampliado y generalizado a *statements* puesto que seguimos la línea de los lenguajes de programación modernos de paradigma imperativo.

El algoritmo base del parser de Pratt es el siguiente:

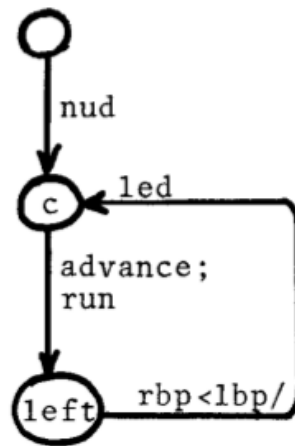


Figura 7: Algoritmo del parser de Pratt

Algorithm 1 Parser de Pratt

```
1: function parse(rbp = 0)
2:   Sea token el token actual.
3:   t ← token
4:   token ← next() ▷ next() devuelve el sgte. token
5:   left ← t.nud()
6:   while rbp < token.lbp do
7:     t ← token
8:     token ← next()
9:     left ← t.led(left)
10:  end while
11:  return left
12: end function
```

Toda la implementación del *parser* gira en torno al algoritmo ???. La idea que subyace es la siguiente: las expresiones están compuestas, en su forma

más básica, por literales (e.g. **1** ó **a**) y operadores (e.g. **+** ó **-**). Una expresión puede ser, por ejemplo **a + 1**. Las expresiones pueden contener, a su vez, subexpresiones como hijos e.g. **a + 1 * 3**. En este último caso **1*3** es una subexpresión que forma el lado derecho de la expresión **+** completa. Nótese que el operador **+** es el más importante en la última expresión y es, por tanto, la forma descrita, y no otra, como **a** lado izquierdo y **1 * 3** lado derecho de la expresión completa la que se toma como correcta. Esto se debe al concepto de precedencia de operadores, de idéntica naturaleza al manejado en matemáticas en la asociatividad de expresiones. Es evidente, entonces, que ***** tiene mayor precedencia que **+**. La importancia de este concepto en el parser de Pratt radica en que en conjunción con el algoritmo ?? se llega al *parser tree*.

Pratt distingue entre 2 tipos de operadores, los que operan hacia la derecha (no left-context) y los que operan de izquierda a derecha sobre 2 operandos, por lo tanto existen dos funciones elementales en el algoritmo de Pratt: *nud* y *led*. La función *nud* (*null denotation*) se utiliza para aquellos símbolos que operan con el elemento dispuesto a su derecha, por ejemplo al comienzo de una expresión. Solo aquellos símbolos que puedan aparecer al comienzo de una expresión, y no tengan operando a su izquierda tendrán función *nud*, como pueden ser los literales (12, "string", **True**) o los operadores en notación prefija (e.g. +1, -2, **not** False).

La función *led* se atribuye a todos los símbolos que trabaja con los operandos de su derecha e izquierda, lo conocido como notación infija. Nótese que un símbolo puede tener por lo tanto hasta 2 interpretaciones, por ejemplo el símbolo **+** asociado al operador **+** puede utilizarse en notación prefija +15 o en notación infija *a + b*, por lo tanto el operador debe disponer de las 2 funciones *nud* y *led*.

En el siguiente ejemplo la función *led* devuelve la evaluación de una expresión (básica o no). Para poder ilustrar todo lo expuesto hasta este punto se muestra la función *led* del operador **+**.

Algorithm 2 Función led del operador **+**

```

1: function ADDLED(self, left)
2:   return left + parse(10)                                ▷ 10 = bp de +
3: end function

```

La función *nud* de los literales tan solo devuelve su valor, como ya se ha señalado.

A continuación se presentan dos ejemplos que ilustran el funcionamiento del algoritmo.

Ejemplo 1:

Expresión = $3 + 1$

parse("3 + 1")

Traza:

PARSE

3: tokens = [3, +, 1]

5: token = 3

EXPRESSION

3: t = 3

4: token = +

5: left = 3

6: rbp (0) < token.lbp (10) /TRUE

7: t = token (+)

8: token = 1

9: left = t.lcd(left) [*]

ADDLED(left) (left == 3)

2: **EXPRESSION(10)**

3: t = 1

4: token = end

5: left = 1

6: rbp (10) < token.lbp (0)

11: return 1

REGRESO A *

2: return 3 + 1

Ejemplo 2:

Expresión = $3 + 1 * 4$

parse("3 + 1 * 4")

Traza:

PARSE

3: tokens = [3, +, 1, *, 4]

5: token = 3

EXPRESSION

3: t = 3

```

4: token = +
5: left = 3
6: rbp (0) < token.lbp (10) /TRUE
7: t = token (+)
8: token = 1
9: left = t.led(left) [*]
ADDLED(left) (left == 3)
2: EXPRESSION(10)
3: t = 1
4: token = *
5: left = 1
6: rbp (10) < token.lbp (20) /TRUE
7: t = *
8: token = 4
9: left = t.led(left) [**]
PRODLED(left) (left == 1)
EXPRESSION(20)
3: t = 4
4: token = end
5: left = 4
6: rbp (20) < token.lbp (0) /FALSE
11: return left (4)
REGRESO A **
2: return left (1) * 4
REGRESO A *
2: return left (3) + expression(10) (1*4 = 4)
6: rbp (10) < token.lbp (0) /FALSE
12: return left (7)

```

Nótese cómo en el segundo ejemplo, que es un superconjunto del segundo, $1*4$ se trata como una subexpresión de la expresión general, consiguiéndose el efecto de que el operador de mayor precedencia, $*$ atraiga hacia sí el símbolo a su lado izquierdo mediante el algoritmo de Pratt. De este modo siempre se respeta la precedencia de operadores tal y como se definen en Matemáticas. Otra mención importante es que en la formación del *AST*, las funciones *nud* y *led* hacen esencialmente lo mismo, que es desarrollar las ramas pertinentes del árbol. En los ejemplos expuestos la función *led* determina los nodos hijo *first* e hijo *second* que serían en $3+1$ el nodo 3 y el nodo 1 respectivamente.

Otro ejemplo básico para comprender el funcionamiento del parser es el siguiente, donde se describe como funciona para parsear $1 * 2 + 3$

```

// call parse(rbp = 0)
1 _ _ _ _
// NUD(1) -> 1
1 * _ _ _
// LED(1, '*' ) = Tree(1, '*', right=parse(bp('*')))
//               = Tree(1, '*', right=parse(20))
    * _ _ _
    / \
1   parse(20)

// start recursion: parse(20)
// NUD(2) -> 2
2 + _
// 20 < bp('+') ?
// nope, return left = 2

// resolvemos la recursion
    * + _
    / \
1   2
// 0 (rbp actual) < bp('+')?
// yes! continue parsing...
// take LED(left, '+') = Tree(left, '+', right=parse(bp('+')))
//                       = Tree(left, '+', right=parse(10))
        +
        / \
        *   parse(10)
        / \
1   2

// start recursion: parse(10)
// spoiler: return 3

// resolvemos la recursion
        +
        / \
        *   3
        / \
1   2

```

Es interesante también estudiar el mecanismo de detección de errores del parser, ilustrado en la figura 8.

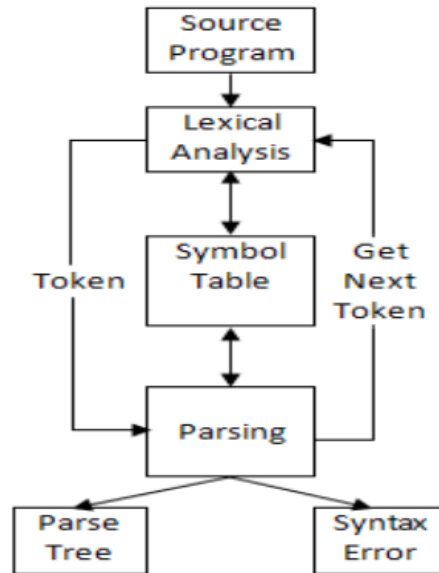


Figura 8: Flujo del parser con detección de errores

4.2.2. Salida

La salida de la implementación del parser descrita en la presente memoria es un árbol en formato de cadena. Los nodos hijos se representan anidados entre paréntesis, recordando a la sintaxis de Lisp, junto con su categoría gramatical. Los argumentos de las funciones se recogen en listas que, aparecen, por tanto entre corchetes. En la tabla 1 se muestran algunos ejemplos de expresiones y statements primitivos, antes de considerar los bloques.

Expresión	Salida Parser
a + b	Add(Name (a),Name (b))
suma(a,b)	FunCall(Name (suma),[[Name (a), Name (b)]])

Cuadro 1: Ejemplos de expresiones y statements primitivos

4.2.3. Análisis de coste

De nuevo, el análisis de coste se torna una tarea complicada, pues en el análisis entran en juego variables como el tipo de token, que condicionan el coste. Es por ello que, de nuevo, se ha optado por una estimación del coste. Se emplea el algoritmo descrito en la sección 4.1.2. El gráfico obtenido es el siguiente:

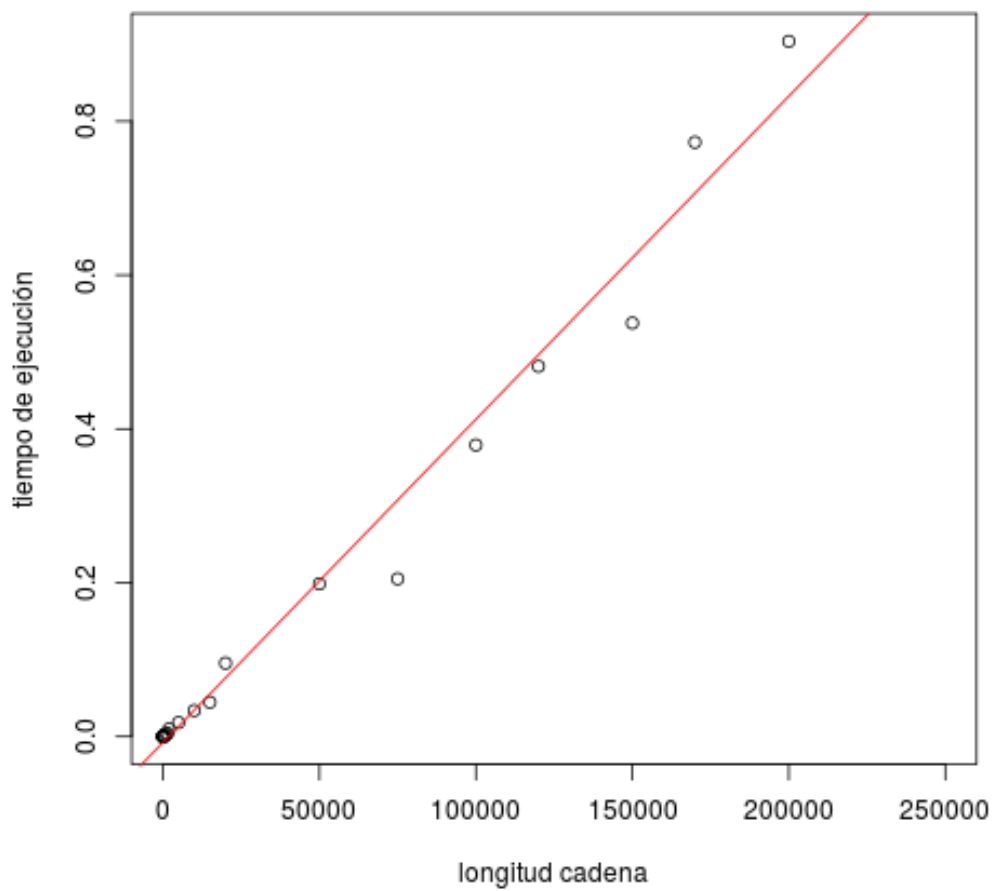


Figura 9: Tiempo de ejecución del parser vs n

Una vez más se aprecia una tendencia lineal (figura 9): el coste de la implementación del parser de Pratt es $\mathcal{O}(n)$, esto se debe a que el algoritmo propuesto por Pratt es una mejora de los parser recursivos descentes y predictivo, por lo tanto debe tener coste $\mathcal{O}(n)$ [19][25]

4.3. Statements

Originalmente el parser de Pratt estaba pensado para lenguajes funcionales como Lisp o CGOL, dado que en este paradigma todo se puede considerar una expresión. Squanchy como la mayoría de los lenguajes modernos más usados (Python, C, Java ...) emplea statements que no se pueden tratar como expresiones. No obstante una de las grandes ventajas del parser es ser independiente a la gramática, por lo tanto podemos añadir nuevos símbolos para representarlos. Primero definimos:

- *statement* es una declaración, una expresión, en términos prácticos es cada línea de código.
- *list of statements* es una agrupación de 2 o más statements, se representa como una lista.
- *block* consiste en uno o más *statement* o agrupaciones de ellos.

Ejemplo en Java:

```
{ //block start
int var = 20; //statement 1
var++;      //statement 2
} //block end
```

En este caso *block* esta delimitado por { } y formado por *list of statements*, que a su vez posee *statement* 1 y 2. En nuestro caso tenemos varios símbolos asociados a statements:

- *if-then-else* asociado al if statement, básico en todos los lenguajes.
- *while*, asociado a while statement, similar a Python.
- *function*, realmente el símbolo empleado es (, asociado a la declaración y llamada de funciones.

Estos símbolos cuentan con un método *nud* que permitirá parsear los distintos statements en su *block* y aridad *statement*. Un ejemplo sencillo para entender este procedimiento es el *while statement*:

```
while expression ::
    statement(s)
```


Como se puede apreciar *block* empieza con el operador `::` y termina en *newline* o *end*, no esta delimitado por operadores como `{}` y los statements que lo forman no necesitan finalizar en `;`. De manera que ¿cómo podemos saber que statements son parte de *block* o no? la respuesta es mediante la indentación, al igual que Python utilizamos los símbolos *newline*, *tab* e *indent* para organizar el código y saber que statements forman parte de bloques y cuales no.

Es fundamental entonces saber cuando parar el procedimiento del parser, para ello se definen:

- Los símbolos de fin de statement o fin de línea (*endStmt*), en muchos lenguajes se utiliza `;`, nosotros al igual que python usamos la indentación.
- Los símbolos de fin de bloque (*endBlock*), en el ejemplo anterior de Java se puede apreciar como el bloque termina en `}`; en nuestro caso este depende del símbolo asociado, por ejemplo el bloque *then* termina en *else* o en *EOF*.

Con estos símbolos conseguimos que el parser sin ninguna modificación consiga tratar correctamente los statements, dado que poseen un *lbp* = 0 el parser los ignora, pero nos sirven como "señales".

Los siguientes algoritmos son los encargados de parsear statements para aquellos símbolos que lo necesitan:

Algorithm 3 Statement. Parsea hasta llegar a *endStmt* o *endBlock*.

```

1: function STATEMENT(endStmt, endBlock)
2:   if token.arity = statement then
3:     return token.nud()                                ▷ solve statement
4:   end if
5:   expr ← parse()
6:   if token == endLine, endBlock then                  ▷ llego a end
7:     return expr                                          ▷ expression
8:   end if
9:   advance()                                              ▷ next token
10: end function

```

Algorithm 4 Statement list. Parsea statements hasta llegar a endBlock.

```

1: function STATEMENTLIST(endBlock)
2:   stmt  $\leftarrow$  [] ▷ list of statement
3:   while True do
4:     if token = endBlock then
5:       break ▷ end of parsing
6:     end if
7:     if token = tab then
8:       advance() ▷ skip indentation
9:     end if
10:    stmt[]  $\leftarrow$  statement(endBlock)
11:  end while
12:  return stmt ▷ stmt in the block
13: end function

```

4.4. Scope

El *Scope* es la región del programa donde las variables son definidas y accesibles; un Scope a su vez puede contener otros como explicaremos a continuación. Modelizamos el Scope o Namespace como una clase con métodos para instanciarlo, añadir variables a este, encontrar el valor de una variable y encontrar el padre del scope. En nuestro caso existe un Global Scope que se puede encontrar en el código como *Scope* que modeliza todo el espacio de nombres del programa. Los nombres *Name* se pueden referir a nombres de variables o funciones, y por lo tanto todos estos deben contemplarse en el *Scope*. En el ejemplo del cuadro 2 se puede apreciar el Scope para el código propuesto.

```

a: 5
b: "hola"
c : a + 10

```

Name	Value
a	5
b	"hola"
c	15

Cuadro 2: Ejemplo de Scope

Nótese que el valor de la variable con *Name* c posee el valor de la expresión.

Al asignar un valor a un nombre con el operador `:` se asigna el valor de la expresión a la derecha del operador; por convenio el valor de las constantes (símbolo *Const*) es ellos mismos, el valor de un nombre es el valor que posee en el scope, y el valor de una expresión es el resultado de la misma, por lo tanto primero se resuelve el valor de la expresión.

Las funciones tienen su propio *Scope* dentro del global, esto permite trabajar con las variables pertinentes dentro del bloque de una función sin problemas. Cada función tiene su propio Scope dentro del global *SCOPE*. Antes hemos mencionado que todos los nombres deben figurar en el SCOPE, eso incluye a los nombres de funciones. Los nombres para identificar una función en nuestro caso son la tupla nombre + argumentos, de manera que *suma (a,b)* no es la misma función que *suma (a,b,c)*. Un ejemplo para entender este mecanismo es el siguiente código:

```
a : 5
fib (1) -> 1
fib (2) -> 1
fib(x) -> fib(x-1) + fib(x-2)
suma (a,b) -> a+b
suma (a,b,c) -> a+b-c
```

Como se puede apreciar en el cuadro 3 el valor asociado a la variable *a* es 5, el valor asociado a la función fibonacci tiene varios valores dependiendo de su parámetro; *fib (1)* es un nombre con un valor asociado 1, *fib (2)* es un nombre con un valor asociado 2 y en especial *fib (x)* es un nombre con un valor asociado desconocido *Undefined* esto se debe a que no conocemos el valor de la función, pero debe registrarse en el espacio de nombres igualmente. De igual forma con las otras entradas de la función suma. La función *suma (a,b)* no tiene un valor definido, dado que el valor de la función depende del parámetro *x*, entonces ¿Qué ocurre cuando hacemos la llamada a la función *suma (3,2)*? Se resolvería la función y se devolvería en este caso el valor correspondiente a 3+2, teniendo una nueva entrada en el Scope que sería Name = *suma (3,2)*, value = 5.

Continuando con este ejemplo, imaginemos que en el código posteriormente tenemos *a : suma(4,5)* el nombre *a* tendrá asociado el valor de *suma(4,5)* que está en el Scope, por lo tanto solo necesitamos una operación de acceso $\mathcal{O}(1)$ para saber el valor de *a*.

En resumen, todos los nombres ya sean variables o llamadas a funciones se guardan en el Scope junto a su respectivo valor, esto hace que se requiera de más espacio pero su eficiencia temporal es $\mathcal{O}(1)$. Destacar que esta solución es propia.

Name	Value
a	5
(fib,1)	1
(fib,2)	2
(fib,x)	Undefined
(suma,a,b)	Undefined
(suma,a,b,c)	Undefined

Cuadro 3: Scope correspondiente al código de abajo

¿Cómo se resolvería $fib(3)$? La resolución de las funciones consiste en resolver todas las expresiones en su bloque, hasta obtener el valor especificado en *return*, llegados a este punto es importante consultar el apartado 5 para entender la sintaxis de las funciones. En este caso $fib(3)$ se resolvería de la siguiente forma:

$fib(3)$

1. Se añade al Scope de la función los parámetros con los valores indicados,; en este caso aparece en el Scope de la función (no en el global) la entrada $Name = x$ por $value = 3$
- 2: Se resuelven todas las expresiones en el cuerpo de la función, en este caso es $fib(x - 1) + fib(x - 2)$
 - 2.1: Para resolver $fib(x-1)$ obtenemos el valor de x del Scope de la función y resolvemos la llamada $fib(2)$, en este caso sabemos su valor dado que se encuentra en el Scope global y es 2, de manera que el valor de la llamada $fib(x - 1)$ es 1, esto no es necesario registrarlo en el Scope.
 - 2.2: Para resolver $fib(x-2)$ se procede como en el paso 2.1 y obtenemos que el valor a la llamada $fib(x - 2)$ es 1.
- 3: El cuerpo de la función por lo tanto sería $1+1$, resolvemos esta expresión básica de aridad 2 y tenemos 2.
- 4: Registramos en el Scope el resultado añadiendo la entrada $Name = (fib, 3)$ y $value = 2$

Respecto a las funciones, en el Scope solo guardamos aquellas llamadas a funciones indicadas explícitamente por el usuario en el código, ya sea porque hace la llamada explícitamente $suma(4, 5)$ o ha definido el resultado como en:

`fib(1)->1`

Otro aspecto importante es la detección de errores, tal y como esta modelizado el Scope podemos tratar 2 tipos de errores esencialmente:

- Si se llama una función cuyo nombre no esta registrado en el scope es inmediato comprobar que en la implementación surgiría un *KeyError* dado que ese nombre no esta en el diccionario. Si eso sucede devolvemos un error como Python: "*name 'nombre de la funcion' not defined*". De igual forma si resolvemos una expresión cualquiera, por ejemplo $a+b$ y una de las variables no esta en el Scope, se devolvería el mismo error.
- Si llamamos dentro de una función a una variable declarada fuera, podemos comprobarlo facilmente al disponer del global scope y del scope de la función, y devolveríamos error "*name 'nombre de la variable' before assignment*".

4.5. Algunos detalles de la implementación

Antes de continuar con las disquisiciones sobre la implementación, es conveniente recordar que el paradigma empleado ha sido el de la orientación al objeto, pues los tokens se presentan como entidades fácilmente representables como objetos, que en el *parser tree* constituirán nodos. Se ha visto en la sección 4.2 que todos los tokens disponen de al menos una de las funciones fundamentales. Cada símbolo se modeliza con una clase que recoge las funciones correspondientes a dicho símbolo y su *lbp*. La función correspondiente al algoritmo 4 se asocia con el símbolo $+$, de modo que la función que parsea expresiones (algoritmo 1) pueda llamarla. Dada esta generalidad para cada símbolo, es conveniente la definición de una protoclasa que sea capaz de generar estas clases de forma dinámica: clase **symbol** en la implementación (*parser.py*).

Estas clases se guardan en un tabla de símbolos (**symbol.table** en la implementación) que permiten acceder a las funciones fundamentales de los símbolos en tiempo $\mathcal{O}(1)$ (para esto se ha elegido implementar la tabla mediante un diccionario).

La generalización a árboles implica la necesidad de referenciar los nodos hijos de cada *token*. Es por esto que se añaden los atributos **first**, **second** y **third**, que apuntarán a los hijos correspondientes en función de la aridad del símbolo, e.g. el símbolo $+$ tendrá dos hijos en notación infija, pero tan solo uno en notación prefija.

4.6. Algunos ejemplos más avanzados

A continuación se muestran algunos ejemplos de la salida del parser. Se acompañan de su representación gráfica correspondiente elaborada con nuestro propio módulo de visualización.

```
while a < 342 ::  
    b = 5  
    a = a + 1
```

El *AST* sería:

```
while(<(Name (a),Const (342)),  
      [Assign(Name (b),Const (5)),  
        =(Name (a),Add(Name (a),Const (1)))])
```

Se puede apreciar que el primer término es la expresión $a < 342$ y el segundo término es una lista de los statements que conforman el while-block: la asignación $b : 5$ y la asignación $a : a + 1$.

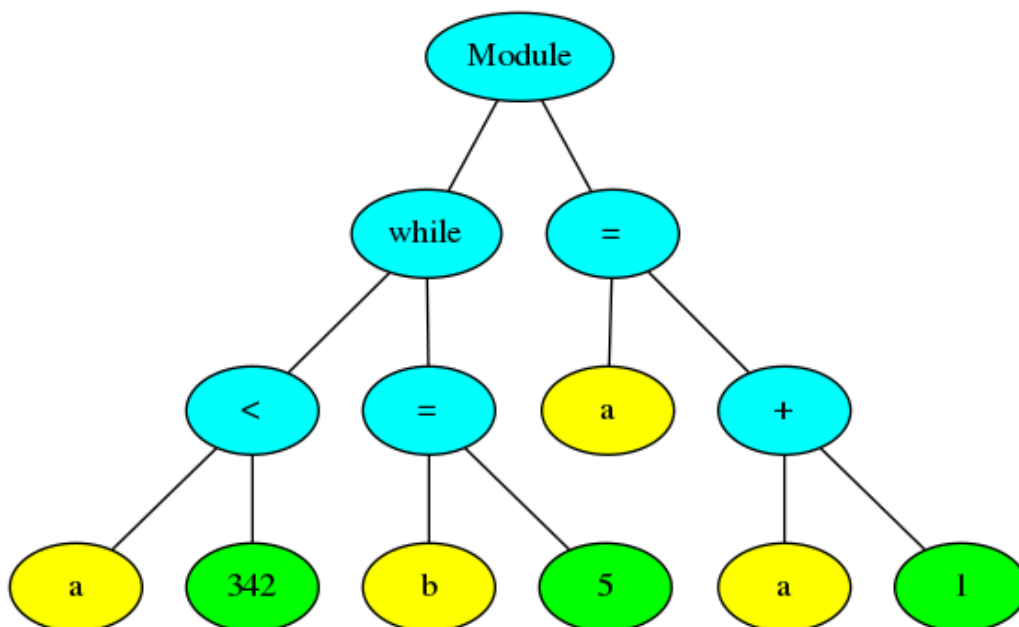


Figura 10: Ejemplo de statement while

```
if b<5 and c>10 then d:45
```

El *AST* sería:

```
IfExp(And(<(Name (b),Const (5)),>(Name (c),Const (10))),  
      [Assign(Name (d),Const (45))])
```

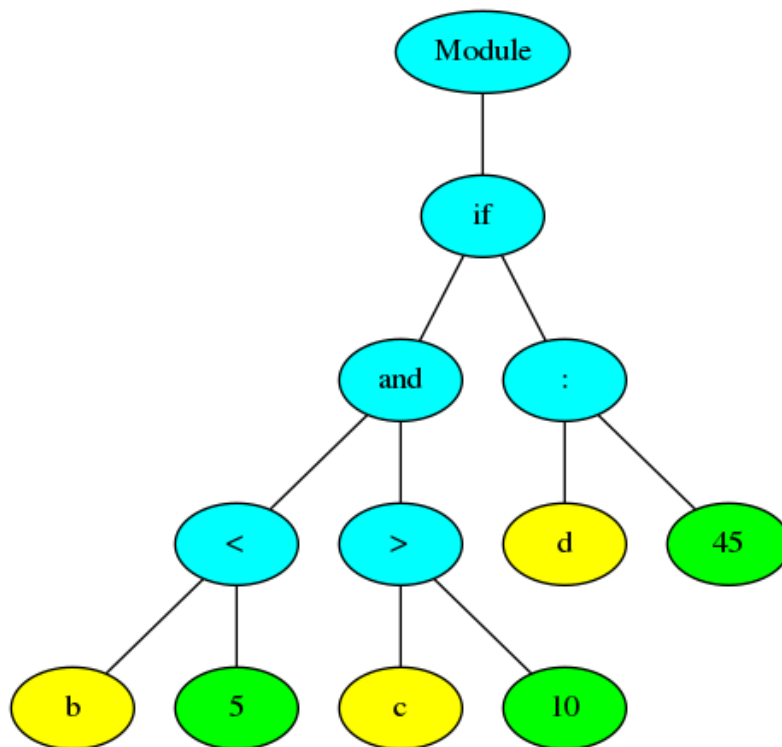


Figura 11: Ejemplo de statement if

5. Sintaxis de SquanchyPL

En esta sección se presenta la sintaxis de SquanchyPL y algunos ejemplos en SquanchyPL que el lector puede someter al *parser* si así lo desea para observar los resultados como cadena de texto o bien utilizar la herramienta de visualización. Se muestra, además, el código en Haskell para establecer comparaciones.

Tipos en SquanchyPL

- **Int** equivale a **int** en C. Almacena números enteros.
- **Double** equivale a **double** en C.
- **List** equivale a las listas de Python o Haskell.
- **Bool** toma los valores 0 ó 1. Existen las palabras clave *True* y *False*.
- **String** no es una tipo de dato primitivo. Es en realidad una lista de caracteres.

Operadores En general, los operadores en SquanchyPL funcionan igual que en otros lenguajes. Dispone de todos los esperables con el orden de precedencia típico. Estas son las únicas diferencias importantes entre los operadores de SquanchyPL y los de Python:

- El operador de asignación es **:** en lugar de **=**.
- El operador de igualdad es **=** en lugar de **==**.

Ejemplos:

```
not a and b
+1 -(-1-5/2)*2**2+(10%2)*3
x * 64 = x << 6
```

En resumen:

- Operadores aritméticos: **+** **-** ***** **/** ****** **%**
- Operadores de comparación: **=** **!=** **<** **>** **<=** **>=**
- Operadores de asignación: **:**
- Operadores lógicos: **and** **or** **not** **<<** **>>**

Variables Una variable es una referencia a un lugar de la memoria que puede almacenar un valor, una cadena, una lista, una función (como en JavaScript). SquanchyPL es de tipado dinámico.


```
myVarName: 78
myVarName: "hello"
myList: [1,2,34,5.6,"hi!",23]
myTuple: (a,b,c)
```

JavaScript:

```
var foo = function(a,b) { return a+b; }
```

SquanchyPL:

```
foo : suma(a,b) -> a+b
```

Listas Basadas en Haskell. Están delimitadas por corchetes ([]) y sus elementos están separados por comas. No hay restricción de tipos, pero los elementos no pueden ser expresiones, han de ser literales de cualquier tipo. Se puede acceder a los elementos con el operador .. A continuación un ejemplo:

Haskell:

```
let numbers = [1,2,3,4]
let truths = [True, False, False]
let strings = ["here", "are", "some", "strings"]
```

SquanchyPL:

```
numbers : [1,2,3,4]
truths : [True, False, False]
strings : ["here", "are", "some", "strings"]

list_of_lists : [1,2,3,[1,2,3]]

mylist : [12,45463,1.56,"hello",45,35,47]

print ( mylist.0)
print (mylist.3)
```

Out:

```
> 12
> "hello"
```

Tuplas Basadas en Haskell. Contienen la combinación de varias expresiones separadas por comas. Se puede acceder a los elementos con el operador `.` como con las **listas**. He aquí un ejemplo:

Haskell

```
(True, 1)
("Hello world", False)
(4, 5, "Six", True, 'b')
```

SquanchyPL

```
my_Tuple: ("Hello world", False)
my_tuple : (1, "hello", 5.6)
print (myTuple.3)
print (myTuple.1)
```

```
Out:
> 5.6
> 1
```

Tuplas anidadas:

```
((2,3), True)
((2,3), [2,3])
[(1,2), (3,4), (5,6)]
```

Globales y constantes SquanchyPL trabaja con espacios de nombres locales (funciones) y globales (módulo).

Una variable **global** es accesible desde cualquier alcance/espacio de nombres. Una variable global no es lo mismo que una variable asignada en alcance global. Las variables globales se pueden declarar explícitamente usando la palabra clave **global** como sigue: **global** var_name. He aquí un ejemplo de uso:

```
global a
a : 5
inc(x) -> x+a
print (inc (5))
print (a)
```

La salida de esto es:

```
> 10
> 5
```

Una variable global solo puede almacenar literales (tipos primitivos). Una **constante** es un valor determinado en tiempo de compilación, inmutable y accesible desde cualquier alcance-espacio de nombres. Las constantes se crean con el operador de asignación de constantes `::`. A continuación un ejemplo de uso y declaración:

```
a := 5
print (a)
inc (x) -> x+a
print (inc (5))
a: 10
```

La salida será:

```
> 5
> 10
> error
```

En este caso **a** ya no es un **Name**, sino un **Const**.

Haskell

```
fib x
| x < 2 = 1
| otherwise = fib (x - 1) + fib (x - 2)

fib 1 = 1
fib 2 = 2
fib x = fib (x - 1) + fib (x - 2)
```

SquanchyPL

```
fib (x) -> y ::
if x<2 then y:1 else y: fib(x-1)+fib(x-2)

fib (1) -> 1
fib (2) -> 2
fib(x) -> fib(x-1) + fib(x-2)
```

Funciones SquanchyPL tiene un enfoque de las funciones similar al del paradigma funcional. La sintaxis básica es:

```
nombre_funcion (arg) -> (return) :: statement(s)
```

De la propia definición de una función deriva la sintaxis de las llamadas a funciones.

```
nombre_funcion (arg)
```

La tupla *arg* se refiere a los argumentos, siempre se tiene que indicar explícitamente aunque la función no reciba argumentos.

```
foo () -> null      # sin argumentos#  
suma (a,b) -> a+b   # argumentos (a,b)#
```

return se refiere a lo que devuelve una función o realiza, no es exactamente igual al `return statement` de Python; este campo siempre debe indicarse, de lo contrario se tratará como una llamada a función. Por lo tanto, no consideramos como tal las funciones que no devuelven nada.

El *return* puede ser un valor (*Const*), una tupla, o una expresión.

```
foo () -> True      # devuelve True #  
suma (a,b) -> a+b   # devuelve la expresión a+b #  
fib (1) -> 1        # devuelve el valor 1  
suma (a,b)          # es la llamada a la función #  
suma (a,b) ->       # error  
suma (a,b) -> (c,d) # devuelve los valores c y d.
```

El cuerpo o bloque de una función es opcional, se empieza con el operador `::` y como se ha explicado en el apartado referido a `statements`, es clave la indentación.

```
suma (a,b) -> (c,d) ::  
  
    c:a+b      #stmt 1  
    d:a-b      #stmt 2  
    print (a)   #stmt 3  
    print (b)   #stmt 4
```

6. LLVM. Generación de código

El módulo encargado de esta tarea es **codegen.py**.

Consideremos como ejemplo la siguiente función en Squanchy.

```
mul_add (x,y,z) -> x*y+z
```

El AST correspondiente es el siguiente:

```
Function(Name (mul_add),[[Name (x), Name (y), Name (z)],  
                        [Add(Mul(Name (x),Name (y)),Name (z))]])
```

Evaluamos el AST, y obtenemos el siguiente código LLVM-IR:

```
define i32 @mul_add(i32 %x, i32 %y, i32 %z) {  
entry:  
    %tmp = mul i32 %x, %y  
    %tmp2 = add i32 %tmp, %z  
    ret i32 %tmp2  
}
```

Este código intermedio se guarda en el fichero *output.ll*, y puede convertirse directamente a código objeto gracias LLVM con el comando *llc*, obteniendo *output.o* de manera que habríamos conseguido compilar ese sencillo programa y para ejecutarlo bastaría con escribir *./output* en la terminal.

Para entender todo el proceso vamos a explicar en más detalle como evaluamos el *AST*, tarea que lleva a cabo el módulo **myeval.py**. Recorremos el *AST* y obtenemos el valor de cada *nodo*, que es el resultado de evaluar la expresión correspondiente a ese nodo y además reflejamos esa expresión en un *IRBuilder* [14] para ir construyendo nuestro IR [21]. Un ejemplo sencillo para entenderlo:

```
4 + 5    # Add(Const (4),Const (5))  
10 - 6   # Sub(Const (10),Const (6))
```

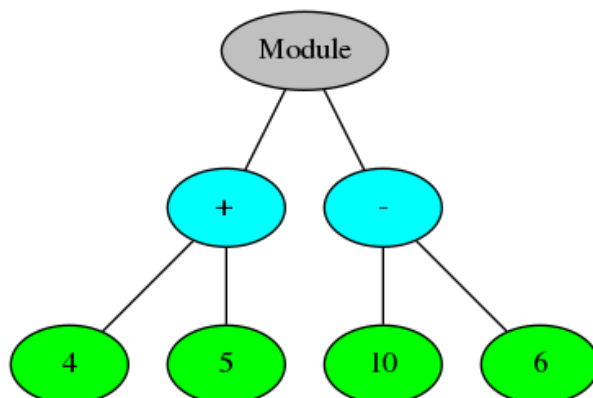


Figura 12: Ejemplo evaluación AST

Recorremos desde *root* (module), y evaluamos los nodos $+$ y $-$; para ello disponemos de un diccionario con operadores y sus respectivas funciones.

```

operations = {
    "+": lambda first,second: first+second,
    "-": lambda first,second: first-second,
    "*": lambda first,second: first*second,
    ...
}

```

En este caso obtenemos el operador $+$, y los hijos *first* = 4 y *second* = 5 del primer nodo correspondiente al símbolo $+$, y devolvemos el resultado de la función correspondiente en el diccionario, que es 9. De igual forma con el segundo nodo correspondiente al operador $-$.

Ahora bien, ¿Cómo aprovechamos la evaluación del AST para ir generando código IR? LLVM nos permite hacerlo gracias a su constructor de código *IRBuilder*. Siguiendo el mismo procedimiento disponemos de otro diccionario equivalente, pero en este caso con la siguiente entrada:

```

"+": lambda first,second: builder.add(first, second)

```

El *builder* es el cuerpo del IR, con la función *add* añadimos al código IR la siguiente línea, muy similar al ensamblador, refleja la suma de esos 2 enteros de 32 bits:

```

%.2" = add i32 4, 5

```

De igual forma con el nodo asociado a la operación — podemos añadir la siguiente introducción al IR:

```
%".3" = sub i32 10, 6
```

La evaluación de asignaciones es similar; al evaluar una asignación creamos un puntero con el nombre de la variable, y almacenamos en este el valor resultante de evaluar el lado de derecho de la asignación, para ello es necesario emplear *alloc* y *store* en el *builder*. Sin embargo, la evaluación de la definición y llamada de funciones es algo más compleja, pero se puede comprobar como se ha realizado en el módulo **myeval.py**.

Destacar que muchas operaciones como *mod* no están disponibles de forma nativa, las tenemos que implementar nosotros, al igual que hemos implementado nuestra propia función *print*. [6]

La mejor parte de usar LLVM no es solo las herramientas que proporciona para facilitar la creación de código intermedio, también nos permite optimizar dicho código empleando técnicas como: asignación de registros, selección de instrucciones, reordenamiento de cálculos, factorización, optimización de bucles ... por lo tanto usamos LLVM tanto en la etapa de generación código como en la de optimización.

El código IR generado se llama *output.ll* basta con hacer **cat output.ll** para consultarlo. Compilar y ejecutar el código en Squanchy se realiza de forma similar a C con gcc:

```
llc -filetype=obj output.ll
clang output.o -o output
./output
```

Para más información sobre LLVM y entender al detalle todo el proceso, recomiendo consultar las referencias [7], [8], [13], [24] para ver algunos ejemplos en los que nos hemos basado y visitar los sitios web oficiales de LLVM, en especial **Kaleidoscope** [15] que consiste en un tutorial realizado por LLVM para el desarrollo de un lenguaje de programación desde cero con dicha herramienta.

En github se puede encontrar el archivo **ir_code**, que contiene un ejemplo completo de código en Squanchy con operaciones aritméticas, asignaciones, funciones ... y su correspondiente código IR y salida.

7. GitHub del proyecto

En github puedes encontrar el código fuente del *parser* y *lexer*. Además puedes encontrar código de ejemplo en *example.md* , *tutorials* con información sobre la gramática y sintaxis. El proyecto a día 5/1/1029 consiste en un compilador para Squanchy que permite la mayoría de operaciones aritméticas, asignación de variables, declaración de funciones sencillas e imprimir resultados por pantalla.

<https://github.com/Jesucrist0/Squanchy-PL>

Referencias

- [1] prattparser. <https://github.com/percolate/pratt-parser>.
- [2] Pratt parsing and precedence climbing are the same algorithm. <https://www.oilshell.org/blog/2016/11/01.html>.
- [3] Review of pratt/tdop parsing tutorials. <https://www.oilshell.org/blog/2016/11/02.html>.
- [4] Alfred V Aho. *Compilers: principles, techniques and tools (for Anna University)*, 2/e. Pearson Education India, 2003.
- [5] Vipin Ajayakumar. Parsing text with python. <https://www.vipinajayakumar.com/parsing-text-with-python/>.
- [6] Marcelo Andrade. Print function on llvmlite. <https://github.com/numba/llvmlite/issues/357>.
- [7] Marcelo Andrade. Writing your own programming language and compiler with python. <https://blog.usejournal.com/writing-your-own-programming-language-and-compiler-with-python-a468970ae6df>.
- [8] Eli Bendersky. Building and using llvmlite - a basic example. <https://eli.thegreenplace.net/2015/building-and-using-llvmlite-a-basic-example/>.
- [9] Biswajit Bhowmik, Abhishek Kumar, Abhishek Kumar Jha, and Rakesh Kumar Agrawal. A new approach of compiler design in context of lexical analyzer and parser generation for nextgen languages. *International Journal of Computer Applications (0975-8887) Volume*, 2010.

- [10] Georg Brandl and Pygments contributors. Write your own lexer. <http://pygments.org/docs/lexerdevelopment/>.
- [11] Douglas Crockford. Top down operator precedence. <http://crockford.com/javascript/tdop/tdop.html>.
- [12] The Python Software Foundation. The python language reference. <https://docs.python.org/3.3/reference/index.html#reference-index>.
- [13] LLVM. Example—defining a simple function. <https://llvmlite.readthedocs.io/en/latest/user-guide/ir/examples.html#>.
- [14] LLVM. Ir builders. <https://llvmlite.readthedocs.io/en/latest/user-guide/ir/ir-builder.html#function-call>.
- [15] LLVM. Kaleidoscope. <https://llvm.org/docs/tutorial/index.html>.
- [16] Fredrik Lundh. Simple top-down parsing in python. <http://effbot.org/zone/simple-top-down-parsing.htm>.
- [17] Fredrik Lundh. Using regular expressions for lexical analysis. <http://effbot.org/zone/xml-scanner.htm>.
- [18] Bob Nystrom. Pratt parsers: Expression parsing made easy. <http://journal.stuffwithstuff.com/2011/03/19/pratt-parsers-expression-parsing-made-easy/>.
- [19] Vaughan R Pratt. Top down operator precedence. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 41–51. ACM, 1973.
- [20] Ruslan Spivak. Let’s build a simple interpreter. <https://ruslanspivak.com/lsbasi-part9/>.
- [21] stackoverflow. Understanding the simplest llvm ir. <https://stackoverflow.com/questions/27447865/understanding-the-simplest-llvm-ir>.
- [22] Federico Tomasetti. A guide to parsing: Algorithms and terminology. <https://tomassetti.me/guide-parsing-algorithms-terminology/>.

- [23] Federico Tomasetti. Parsing in python: Tools and libraries. <https://tomasetti.me/parsing-in-python/>.
- [24] Wikipedia. Llvm. <https://en.wikipedia.org/wiki/LLVM>.
- [25] Wikipedia. Recursive descent parser. https://en.wikipedia.org/wiki/Recursive_descent_parser.
- [26] William W Wold. Pinecone. <https://medium.freecodecamp.org/the-programming-language-pipeline-91d3f449c919>.