UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ciencias, 2026-1

Lenguajes de Programación

Proyecto 1: MINILISP

Cruz Martínez Giovanny 312001690

Guadalupe Abrego Karla Sheridam - 320247314

Villa Briseño Fabián - 318349941

Introducción

1. Introducción

En este proyecto extendemos el lenguaje **MINILISP** con nuevas construcciones que lo hacen más expresivo y útil. La idea es partir de la versión básica que vimos en clase y agregarle operadores, estructuras de datos y formas de control que nos permitan escribir programas más interesantes.

Como vimos en las notas del curso Soto Romero, 2025h, 2025i, un lenguaje de programación se puede especificar formalmente mediante tres componentes principales: su sintaxis concreta (cómo se escriben los programas), su sintaxis abstracta (cómo se representan internamente) y su semántica (qué significan y cómo se evalúan). En este proyecto trabajamos con estos tres niveles para cada una de las extensiones que agregamos. Las extensiones que implementamos son:

- **Operadores variádicos**: Permitimos que operadores como `+`, `*`, `and` y `or` acepten cualquier número de argumentos, no solo dos. Esto hace el lenguaje más flexible y natural de usar.
- **Nuevos operadores aritméticos**: Agregamos multiplicación (`*`), división (`/`), módulo (`mod`) y exponenciación (`expt`) para poder hacer más operaciones matemáticas.
- **Operadores relacionales**: Incluimos comparadores como `<`, `>`, `<=`, `>=` y `=` que también funcionan de forma variádica, permitiendo comparaciones encadenadas como en matemáticas.
- **Pares ordenados**: Implementamos la construcción `cons` para crear pares, junto con `car` y `cdr` para acceder a sus componentes, siguiendo la tradición de LISP Soto Romero, 2025f.
- **Expresiones `let` y `let*` variádicas**: Extendemos las expresiones `let` para permitir declarar múltiples variables a la vez. También agregamos `let*` que permite que cada variable use las anteriores en su definición Soto Romero, 2025d.
- **Condicionales booleanos**: Agregamos `if` para tomar decisiones basadas en condiciones, lo cual es fundamental en cualquier lenguaje de programación.
- **Listas**: Implementamos listas como una estructura de datos recursiva usando pares y el valor especial `empty`, junto con operaciones como `list`, `head` y `tail`.
- **Funciones variádicas**: Extendemos las funciones lambda para que puedan recibir cualquier número de argumentos, haciéndolas más flexibles.

Para cada una de estas implementaciones, seguimos el mismo proceso: primero definimos su sintaxis concreta en notación EBNF, luego su representación como árbol de sintaxis abstracta, después especificamos cómo se desazucaran (si es necesario) a construcciones más simples, y finalmente damos su semántica operacional mediante reglas de transición Soto Romero, 2025e.

El proyecto está implementado en Haskell usando las herramientas Alex para el análisis léxico y Happy para el análisis sintáctico, siguiendo el enfoque visto en clase. La implementación completa incluye un intérprete que evalúa programas en MINILISP extendido y produce sus resultados.

En las siguientes secciones presentamos los detalles técnicos de cada componente del lenguaje, mostrando cómo se integran todas las piezas para formar un lenguaje funcional completo y consistente.

1. Formalización

La formalización es, en esencia, **la hoja de ruta de nuestro lenguaje**. Al hacer esto, convertimos todas las ideas y reglas intuitivas que tenemos para MiniLisp en una especificación rigurosa. Esta especificación es la base, porque nos obliga a tener una **definición precisa y sin ambigüedades**, lo

cual es indispensable para que el intérprete, y programadores, podamos entender y razonar de forma clara sobre cómo se comporta el código (Soto Romero (2025i)). En este capítulo, vamos a definir esas reglas fundamentales de MiniLisp, siguiendo el orden de las etapas que vimos en clase.

2. Sintaxis Léxica

La sintaxis léxica define los tokens básicos del lenguaje, es decir, las unidades mínimas de significado como palabras reservadas, identificadores, números y operadores. Para MINILISP, usamos expresiones regulares para especificar cada tipo de token.

2.1. Alfabeto

El alfabeto de MINILISP incluye:

- Dígitos: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Letras: $\{a, b, \ldots, z, A, B, \ldots, Z\}$
- Operadores: $\{+, -, *, /, <, >, =\}$
- Delimitadores: $\{(,), [,], , \}$
- Símbolos especiales: $\{\#\}$

2.2. Tokens

Los tokens de MINILISP se definen mediante las siguientes expresiones regulares:

Números enteros: Un número puede tener signo negativo opcional y debe empezar con cualquier dígito seguido de más dígitos.

$$\text{INT} = \text{DIGIT}^+ \mid -\text{DIGIT}^+$$

Booleanos: Los valores booleanos se representan con `#t` para verdadero y `#f` para falso.

$$\text{BOOL} = \#t \mid \#f$$

Identificadores: Un identificador empieza con una letra y puede contener letras, dígitos o guiones bajos.

$$\text{ID} = \text{LETTER}(\text{LETTER} \mid \text{DIGIT} \mid \_)^*$$

Palabras reservadas: Las palabras clave del lenguaje incluyen:

```
let, let*, if, if0, cond, else, lambda, fst, snd,
head, tail, add1, sub1, sqrt, expt
```

Operadores: Los operadores aritméticos, relacionales y lógicos son:

```
+, -, *, /, =, <, >, <=, >=, !=, not
```

Delimitadores: Los símbolos que delimitan expresiones:

```
(, ), [, ], ,
```

Espacios en blanco: Los espacios, tabulaciones y saltos de línea se ignoran durante el análisis léxico.

2. 3. Sintaxis Libre de Contexto (EBNF)

La sintaxis libre de contexto de MINILISP se define usando notación EBNF. Esta gramática describe cómo se combinan los tokens para formar expresiones válidas del lenguaje.

3.1. Gramática EBNF

$$
\begin{aligned}
\langle S \rangle &::= \langle \text{Expr} \rangle \\
\\
\langle \text{Expr} \rangle &::= \langle \text{Int} \rangle \\
&\quad | \ \langle \text{Bool} \rangle \\
&\quad | \ \langle \text{Id} \rangle \\
&\quad | \ (\langle \text{Op} \rangle \ \langle \text{Expr} \rangle^{+}) \\
&\quad | \ (\text{fst} \ \langle \text{Expr} \rangle) \\
&\quad | \ (\text{snd} \ \langle \text{Expr} \rangle) \\
&\quad | \ (\text{head} \ \langle \text{Expr} \rangle) \\
&\quad | \ (\text{tail} \ \langle \text{Expr} \rangle) \\
&\quad | \ (\text{add1} \ \langle \text{Expr} \rangle) \\
&\quad | \ (\text{sub1} \ \langle \text{Expr} \rangle) \\
&\quad | \ (\text{sqrt} \ \langle \text{Expr} \rangle) \\
&\quad | \ (\text{expt} \ \langle \text{Expr} \rangle \ \langle \text{Expr} \rangle) \\
&\quad | \ (\text{not} \ \langle \text{Expr} \rangle) \\
&\quad | \ (\langle \text{Expr} \rangle, \langle \text{Expr} \rangle) \\
&\quad | \ [\langle \text{Expr} \rangle^{*}] \\
&\quad | \ (\text{let} \ ((\langle \text{Id} \rangle \ \langle \text{Expr} \rangle)^{+}) \ \langle \text{Expr} \rangle) \\
&\quad | \ (\text{let*} \ ((\langle \text{Id} \rangle \ \langle \text{Expr} \rangle)^{+}) \ \langle \text{Expr} \rangle) \\
&\quad | \ (\text{if} \ \langle \text{Expr} \rangle \ \langle \text{Expr} \rangle \ \langle \text{Expr} \rangle) \\
&\quad | \ (\text{if0} \ \langle \text{Expr} \rangle \ \langle \text{Expr} \rangle \ \langle \text{Expr} \rangle) \\
&\quad | \ (\text{cond} \ \langle \text{Clause} \rangle^{+} \ (\text{else} \ \langle \text{Expr} \rangle)) \\
&\quad | \ (\text{lambda} \ (\langle \text{Id} \rangle^{*}) \ \langle \text{Expr} \rangle) \\
&\quad | \ (\langle \text{Expr} \rangle \ \langle \text{Expr} \rangle^{*}) \\
\\
\langle \text{Clause} \rangle &::= [\langle \text{Expr} \rangle \ \langle \text{Expr} \rangle] \\
\\
\langle \text{Op} \rangle &::= + \mid - \mid * \mid / \mid = \mid < \mid > \mid \leq \mid \geq \mid \neq \\
\\
\langle \text{Int} \rangle &::= \text{INT} \\
\langle \text{Bool} \rangle &::= \text{BOOL} \\
\langle \text{Id} \rangle &::= \text{ID}
\end{aligned}
$$

3.2. Notas sobre la gramática

- Los operadores aritméticos y relacionales son variádicos, aceptan dos o más argumentos.

- Las listas se escriben con corchetes y pueden estar vacías.
- `let` y `let*` permiten múltiples asignaciones.
- `cond` requiere al menos una cláusula y debe terminar con `else`.
- Las lambdas pueden tener cero o más parámetros.
- La aplicación de funciones se escribe como `(f arg1 arg2 ...)`.

4. Sintaxis Abstracta

La sintaxis abstracta representa la estructura lógica de los programas sin los detalles superficiales de la sintaxis concreta. Definimos los Árboles de Sintaxis Abstracta (ASA) mediante reglas de inferencia.

4.1. Definición de ASA

Definimos la relación $e \in \text{ASA}$ que se lee como "e es un ASA". Las reglas son:

Números:

$$\frac{n \in \mathbb{Z}}{\text{NumS}(n) \in \text{ASA}}$$

Booleanos:

$$\frac{b \in \{\text{true}, \text{false}\}}{\text{BoolS}(b) \in \text{ASA}}$$

Identificadores:

$$\frac{x \in \text{Id}}{\text{IdS}(x) \in \text{ASA}}$$

Operadores binarios:

$$\frac{e_1 \in \text{ASA} \quad e_2 \in \text{ASA}}{\text{OpS}(\text{op}, [e_1, e_2, \ldots]) \in \text{ASA}}$$

donde $\text{op} \in \{+, -, *, /, =, <, >, \leq, \geq, \neq\}$ y la lista puede tener dos o más elementos.

Operadores unarios:

$$\frac{e \in \text{ASA}}{\text{UnOpS}(\text{op}, e) \in \text{ASA}}$$

donde $\text{op} \in \{\text{not}, \text{add1}, \text{sub1}, \text{sqrt}, \text{fst}, \text{snd}, \text{head}, \text{tail}\}$.

Potencia:

$$\frac{e_1 \in \text{ASA} \quad e_2 \in \text{ASA}}{\text{ExptS}(e_1, e_2) \in \text{ASA}}$$

Pares:

$$\frac{e_1 \in \text{ASA} \quad e_2 \in \text{ASA}}{\text{PairS}(e_1, e_2) \in \text{ASA}}$$

Listas:

$$\frac{e_1 \in \text{ASA} \quad \ldots \quad e_n \in \text{ASA}}{\text{ListS}([e_1, \ldots, e_n]) \in \text{ASA}}$$

Let:

$$\frac{x_1, \ldots, x_n \in \text{Id} \quad e_1, \ldots, e_n \in \text{ASA} \quad e \in \text{ASA}}{\text{LetS}([(x_1, e_1), \ldots, (x_n, e_n)], e) \in \text{ASA}}$$

Let*:

$$\frac{x_1, \ldots, x_n \in \text{Id} \quad e_1, \ldots, e_n \in \text{ASA} \quad e \in \text{ASA}}{\text{LetRecS}([(x_1, e_1), \ldots, (x_n, e_n)], e) \in \text{ASA}}$$

If:

$$\frac{e_1 \in \text{ASA} \quad e_2 \in \text{ASA} \quad e_3 \in \text{ASA}}{\text{IfS}(e_1, e_2, e_3) \in \text{ASA}}$$

If0:

$$\frac{e_1 \in \text{ASA} \quad e_2 \in \text{ASA} \quad e_3 \in \text{ASA}}{\text{If0S}(e_1, e_2, e_3) \in \text{ASA}}$$

Cond:

$$\frac{e_1, \ldots, e_n, e'_1, \ldots, e'_n, e \in \text{ASA}}{\text{CondS}([(e_1, e'_1), \ldots, (e_n, e'_n)], e) \in \text{ASA}}$$

Lambda:

$$\frac{x_1, \ldots, x_n \in \text{Id} \quad e \in \text{ASA}}{\text{LamS}([x_1, \ldots, x_n], e) \in \text{ASA}}$$

Aplicación:

$$\frac{e_1, \ldots, e_n \in \text{ASA}}{\text{AppS}(e_1, [e_2, \ldots, e_n]) \in \text{ASA}}$$

5. Desugaring (Desazucarado)

El desugaring o desazucarado es una técnica en el diseño de lenguajes de programación que consiste en traducir construcciones sintácticas complejas a otras más simples que ya tienen semántica definida Aho et al., 2008. Esto nos permite agregar nuevas características al lenguaje sin tener que definir su semántica operacional desde cero, simplemente las traducimos a construcciones que ya sabemos cómo evaluar.

En nuestro proyecto, usamos desugaring para varias construcciones que pueden expresarse en términos de otras más básicas. Esto simplifica tanto la especificación formal del lenguaje como su implementación.

5.1. Operadores Variádicos

Los operadores variádicos (que aceptan cualquier número de argumentos) se desazucaran a aplicaciones anidadas del operador binario correspondiente. Por ejemplo:

- `(+ e)` $\rightsquigarrow$ `e`
- `(+ e e)` $\rightsquigarrow$ `(+ e e)`
- `(+ e e e ... e)` $\rightsquigarrow$ `(+ e (+ e (+ e ... e)))`

Esto aplica para `+`, `*`, `and` y `or`. El caso especial es cuando no hay argumentos:

- `(+)` $\rightsquigarrow$ `0`
- `(*)` $\rightsquigarrow$ `1`
- `(and)` $\rightsquigarrow$ `#t`
- `(or)` $\rightsquigarrow$ `#f`

Estos son los elementos neutros de cada operación.

5.2. Operadores Relacionales Variádicos

Los operadores relacionales como `<`, `>`, `<=`, `>=` y `=` también son variádicos y permiten comparaciones encadenadas. Se desazucaran a conjunciones de comparaciones binarias:

- `(<e e)` $\rightsquigarrow$ `(<e e)`
- `(<e e e)` $\rightsquigarrow$ `(and (<e e) (<e e))`
- `(<e e ... e)` $\rightsquigarrow$ `(and (<e e) (<e e) ... (<e e))`

Esto permite escribir expresiones como `(<1 x 10)` que se lee naturalmente como "1 es menor que x y x es menor que 10", igual que en matemáticas.

5.3. Expresiones Let Variádicas

Las expresiones `let` con múltiples ligaduras se desazucaran a `let`s anidados:

```
(let ([x e] [x e] ... [x e]) body)
```

$\rightsquigarrow$

```
(let ([x e]) (let ([x e]) ... (let ([x e]) body)))
```

Es importante notar que en `let` las expresiones `e, e, ..., e` se evalúan en el ambiente original, sin poder usar las variables declaradas anteriormente en el mismo `let` Soto Romero, 2025d.

5.4. Expresiones Let* Variádicas

Las expresiones `let*` son similares pero permiten que cada expresión use las variables declaradas anteriormente. Se desazucaran exactamente igual que `let` variádico:

```
(let* ([x e] [x e] ... [x e]) body)
```

$\rightsquigarrow$

```
(let ([x e]) (let ([x e]) ... (let ([x e]) body)))
```

La diferencia con `let` es que aquí cada `e` puede usar las variables `x, ..., x` porque se evalúa en un ambiente que ya las incluye.

5.5. Listas

La construcción `list` para crear listas se desazucara a aplicaciones anidadas de `cons`:

- `(list)` $\rightsquigarrow$ `empty`
- `(list e)` $\rightsquigarrow$ `(cons e empty)`
- `(list e e ... e)` $\rightsquigarrow$ `(cons e (cons e ... (cons e empty)))`

Esto muestra que las listas son simplemente una forma conveniente de escribir estructuras de pares anidados que terminan en `empty`.

Las operaciones `head` y `tail` se desazucaran a `car` y `cdr`:

- `(head e)` $\rightsquigarrow$ `(car e)`
- `(tail e)` $\rightsquigarrow$ `(cdr e)`

5.6. Funciones Variádicas

Las funciones lambda con múltiples parámetros se desazucaran a lambdas anidadas (currying):

```
(lambda (x x ... x) body)
```

$$\rightsquigarrow$$

```
(lambda (x) (lambda (x) ... (lambda (x) body)))
```

De manera correspondiente, las aplicaciones de funciones con múltiples argumentos se desazucaran a aplicaciones anidadas:

```
(f e e ... e)
```

$$\rightsquigarrow$$

```
(((f e) e) ... e)
```

Esto sigue el enfoque del Cálculo Lambda Soto Romero, 2025c, donde todas las funciones son unarias y las funciones de múltiples argumentos se simulan mediante funciones que retornan funciones.

5.7. Ventajas del Desugaring

En nuestra implementación, el desugaring se realiza como una fase separada entre el parsing y la evaluación, transformando el ASA extendido en un ASA del núcleo del lenguaje que luego es interpretado por las reglas de semántica operacional.

5. 6. Semántica Operacional

La semántica operacional describe cómo se evalúan las expresiones del lenguaje paso a paso. Como vimos en las notas del curso Soto Romero, 2025e, usaremos semántica estructural (o de paso pequeño) que modela la ejecución mediante una relación de transición $\rightarrow$ entre estados del programa.

Un sistema de transición para nuestro lenguaje se define como una tupla (E, δ, I, F) donde:

- E es el conjunto de estados (expresiones en sintaxis abstracta)
- δ es la relación de transición que describe cómo evoluciona cada expresión
- I es el conjunto de estados iniciales (cualquier expresión bien formada)
- F es el conjunto de estados finales (valores que no se pueden reducir más)

Los valores finales en nuestro lenguaje son: números, booleanos, pares, listas, el valor `empty` y closures (funciones con su ambiente capturado).

A continuación presentamos las reglas de transición para cada construcción del lenguaje. Usamos la notación $e \rightarrow e'$ para indicar que la expresión e se reduce en un paso a la expresión e'.

6.1. Valores

Los valores no se reducen más, son estados finales:

$$\texttt{Num}(n) \rightarrow \texttt{Num}(n)$$

$$\texttt{Boolean}(b) \rightarrow \texttt{Boolean}(b)$$

$$\texttt{Empty} \rightarrow \texttt{Empty}$$

6.2. Operadores Aritméticos Binarios

Para la suma, tenemos tres reglas según el estado de los operandos:

$$\frac{i \to i'}{\texttt{Add}(i, d) \to \texttt{Add}(i', d)}$$

$$\frac{d \to d'}{\texttt{Add}(\texttt{Num}(n_1), d) \to \texttt{Add}(\texttt{Num}(n_1), d')}$$

$$\texttt{Add}(\texttt{Num}(n_1), \texttt{Num}(n_2)) \to \texttt{Num}(n_1 + n_2)$$

Las reglas para resta, multiplicación, división, módulo y exponenciación son análogas. Por ejemplo, para la multiplicación:

$$\frac{i \to i'}{\texttt{Mul}(i, d) \to \texttt{Mul}(i', d)}$$

$$\frac{d \to d'}{\texttt{Mul}(\texttt{Num}(n_1), d) \to \texttt{Mul}(\texttt{Num}(n_1), d')}$$

$$\texttt{Mul}(\texttt{Num}(n_1), \texttt{Num}(n_2)) \to \texttt{Num}(n_1 \times n_2)$$

6.3. Operadores Relacionales

Los operadores relacionales se evalúan de forma similar. Para el operador menor que:

$$\frac{i \to i'}{\texttt{Lt}(i, d) \to \texttt{Lt}(i', d)}$$

$$\frac{d \to d'}{\texttt{Lt}(\texttt{Num}(n_1), d) \to \texttt{Lt}(\texttt{Num}(n_1), d')}$$

$$\texttt{Lt}(\texttt{Num}(n_1), \texttt{Num}(n_2)) \to \texttt{Boolean}(n_1 < n_2)$$

6.4. Operadores Lógicos

Para la negación:

$$\frac{e \to e'}{\texttt{Not}(e) \to \texttt{Not}(e')}$$

$$\texttt{Not}(\texttt{Boolean}(b)) \to \texttt{Boolean}(\neg b)$$

Para la conjunción:

$$\frac{i \to i'}{\texttt{And}(i, d) \to \texttt{And}(i', d)}$$

$$\frac{d \to d'}{\texttt{And}(\texttt{Boolean}(b_1), d) \to \texttt{And}(\texttt{Boolean}(b_1), d')}$$

$$\texttt{And}(\texttt{Boolean}(b_1), \texttt{Boolean}(b_2)) \to \texttt{Boolean}(b_1 \wedge b_2)$$

6.5. Condicionales

El condicional `if` evalúa primero la condición y luego elige una rama:

$$\frac{c \to c'}{\texttt{If}(c, t, e) \to \texttt{If}(c', t, e)}$$

$$\texttt{If}(\texttt{Boolean}(\texttt{true}), t, e) \to t$$

$$\texttt{If}(\texttt{Boolean}(\texttt{false}), t, e) \to e$$

6.6. Pares Ordenados

Para construir un par, primero evaluamos ambos componentes:

$$\frac{f \to f'}{\texttt{Cons}(f, s) \to \texttt{Cons}(f', s)}$$

$$\frac{s \to s'}{\texttt{Cons}(v, s) \to \texttt{Cons}(v, s')}$$

donde v es un valor. Un par con ambos componentes evaluados es un valor final.
Para acceder a los componentes:

$$\frac{e \to e'}{\texttt{Car}(e) \to \texttt{Car}(e')}$$

$$\texttt{Car}(\texttt{Cons}(f, s)) \to f$$

$$\frac{e \to e'}{\texttt{Cdr}(e) \to \texttt{Cdr}(e')}$$

$$\texttt{Cdr}(\texttt{Cons}(f, s)) \to s$$

6.7. Listas

Las operaciones sobre listas se definen en términos de pares y `empty`:

$$\frac{e \to e'}{\texttt{Head}(e) \to \texttt{Head}(e')}$$

$$\texttt{Head}(\texttt{Cons}(h, t)) \to h$$

$$\frac{e \to e'}{\texttt{Tail}(e) \to \texttt{Tail}(e')}$$

$$\texttt{Tail}(\texttt{Cons}(h, t)) \to t$$

6.8. Expresiones Let

Para las expresiones `let`, evaluamos el valor y luego sustituimos en el cuerpo Soto Romero, 2025d:

$$\frac{v \to v'}{\texttt{Let}(x, v, c) \to \texttt{Let}(x, v', c)}$$

$$\texttt{Let}(x, v, c) \to c[x := v]$$

donde $c[x := v]$ denota la sustitución de todas las ocurrencias libres de x en c por v.

6.9. Funciones

Las funciones lambda con su ambiente forman closures que son valores:

$$\texttt{Closure}(\texttt{params}, \texttt{body}, \rho) \to \texttt{Closure}(\texttt{params}, \texttt{body}, \rho)$$

Para la aplicación de funciones, primero evaluamos el operador y luego los argumentos:

$$\frac{f \to f'}{\texttt{App}(f, \texttt{args}) \to \texttt{App}(f', \texttt{args})}$$

$$\frac{\texttt{args}_i \to \texttt{args}'_i}{\texttt{App}(\texttt{Closure}(...), \texttt{args}) \to \texttt{App}(\texttt{Closure}(...), \texttt{args}')}$$

donde $\texttt{args}'_i$ indica que se evalúa el primer argumento no evaluado.

Cuando todos los argumentos están evaluados, extendemos el ambiente del closure con los parámetros ligados a los argumentos y evaluamos el cuerpo:

$$\texttt{App}(\texttt{Closure}(\texttt{params}, \texttt{body}, \rho), \texttt{vals}) \to \texttt{body}[\rho']$$

donde ρ' es el ambiente ρ extendido con las ligaduras de `params` a `vals`.

Resultados

7. Resultados

La implementación de MINILISP se realizó en Haskell usando Alex para el análisis léxico y Happy para el análisis sintáctico. El intérprete sigue el pipeline:

$$\text{interp} \circ \text{desugar} \circ \text{parser} \circ \text{lexer}$$

7.1. Estructura del proyecto

El proyecto se organiza en los siguientes módulos:

- `Lex.x`: Especificación del analizador léxico en Alex
- `Grammars.y`: Especificación del parser en Happy
- `MiniLisp.hs`: Definición de tipos de datos para ASA y valores
- `Desugar.hs`: Implementación de la desazucarización
- `Interp.hs`: Implementación del intérprete

7.2. Compilación

Para compilar el proyecto:

```
alex Lex.x
happy Grammars.y
ghc --make MiniLisp.hs -o minilisp
```

7.3. Ejemplos de ejecución

Suma de los primeros n naturales:

```
(let ((sum-n (lambda (n)
    (if (= n 0)
        0
        (+ n (sum-n (- n 1)))))))
  (sum-n 10))
=> 55
```

Factorial:

```
(let ((fact (lambda (n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))))
  (fact 5))
=> 120
```

Fibonacci:

```
(let ((fib (lambda (n)
    (if (< n 2)
        n
        (+ (fib (- n 1)) (fib (- n 2)))))))
  (fib 8))
=> 21
```

Map:

```
(let ((map (lambda (f lst)
    (if (= lst [])
        []
        [(f (head lst)) (map f (tail lst))]))))
  (map (lambda (x) (* x 2)) [1 2 3 4]))
=> [2 4 6 8]
```

Filter:

```
(let ((filter (lambda (p lst)
    (if (= lst [])
        []
        (if (p (head lst))
            [(head lst) (filter p (tail lst))]
            (filter p (tail lst)))))))
  (filter (lambda (x) (> x 2)) [1 2 3 4 5]))
=> [3 4 5]
```

7.4. Pruebas

Se implementaron pruebas para verificar el correcto funcionamiento de todas las extensiones:

- Operadores variádicos
- Operadores relacionales encadenados
- Pares y proyecciones
- Listas y operaciones sobre listas
- Let y let* variádicos
- Condicionales if, if0 y cond
- Lambdas y aplicaciones variádicas

Todas las pruebas pasaron exitosamente, confirmando que la implementación es correcta y completa.

Conclusiones

8. Conclusiones

En este proyecto logramos extender el lenguaje MINILISP con nuevas construcciones que lo hacen mucho más completo y útil para escribir programas reales. A lo largo del desarrollo, aplicamos los conceptos teóricos vistos en la clases sobre sintaxis, semántica y diseño de lenguajes de programación.

El proceso de especificar formalmente cada extensión nos permitió entender mejor cómo funcionan estas construcciones en lenguajes reales como Racket o Haskell. Definir la sintaxis concreta en EBNF, traducirla a sintaxis abstracta, y luego especificar su semántica operacional mediante reglas de transición nos dio una visión completa de lo que significa agregar una nueva característica a un lenguaje Soto Romero, 2025e, 2025h, 2025i.

Una de las cosas más importantes fue entender el poder del desugaring. Al traducir construcciones complejas a otras más simples, pudimos mantener el núcleo del lenguaje pequeño y manejable, mientras que la sintaxis superficial ofrece muchas comodidades al programador Aho et al., 2008. Esto es exactamente lo que hacen los lenguajes de programación reales: ofrecen una sintaxis rica y expresiva que internamente se traduce a un conjunto pequeño de operaciones primitivas.

La implementación en Haskell usando Alex y Happy nos permitió ver cómo se construye un intérprete real, desde el análisis léxico y sintáctico hasta la evaluación. Trabajar con estas herramientas nos dio experiencia práctica en el uso de generadores de analizadores, que son fundamentales en la construcción de compiladores e intérpretes.

Las extensiones que implementamos cubren aspectos fundamentales de los lenguajes de programación:

- Los **operadores variádicos** muestran cómo la sintaxis puede ser más flexible que la semántica, usando desugaring para traducir múltiples argumentos a operaciones binarias.
- Los **pares y listas** nos enseñaron sobre estructuras de datos recursivas y cómo se pueden construir estructuras complejas a partir de primitivas simples, siguiendo la estructura de LISP.
- Las **expresiones let y let*** nos mostraron la importancia del alcance de variables y cómo diferentes estrategias de ligadura afectan el comportamiento del programa Soto Romero, 2025a, 2025d.

- Los **condicionales** son esenciales para cualquier lenguaje y su implementación nos mostró cómo el control de flujo se modela en la semántica operacional.
- Las **funciones variádicas** y su desugaring a funciones currificadas nos permiten entender el Cálculo Lambda y nos mostraron cómo las funciones de múltiples argumentos son en realidad azúcar sintáctica Soto Romero, 2025c.

Un aspecto importante que aprendimos fue la necesidad de ser cuidadosos con los detalles. Pequeños errores en la definición de las reglas de transición o en el algoritmo de sustitución pueden llevar a comportamientos incorrectos o incluso a que el intérprete entre en ciclos infinitos. Esto nos enseñó la importancia de la especificación formal y de las pruebas exhaustivas.

También nos dimos cuenta de que hay muchas decisiones de diseño involucradas en la creación de un lenguaje. Por ejemplo, decidir si los operadores relacionales deben ser variádicos, cómo manejar la división por cero, o si las funciones deben evaluarse con estrategia glotona o perezosa. Cada decisión tiene implicaciones en la expresividad, eficiencia y facilidad de uso del lenguaje.

Finalmente, este trabajo nos preparó mejor para entender lenguajes de programación más complejos y para apreciar las decisiones de diseño que tomaron sus creadores. También nos dio las herramientas conceptuales y prácticas para, en el futuro, diseñar nuestros propios lenguajes o extender lenguajes existentes según nuestras necesidades.

En resumen, el proyecto fue una experiencia valiosa que integró teoría y práctica, mostrándonos cómo los conceptos abstractos de la teoría de lenguajes de programación se materializan en implementaciones concretas y funcionales.

Bibliografía

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2008). *Compiladores. Principios, técnicas y herramientas* (Segunda edición). Pearson Educación.

Harper, R. (2012). *Practical Foundations for Programming Languages* [Version 1.32 of 05.15.2012]. Consultado el 15 de septiembre de 2025, desde http://www.cs.cmu.edu/~rwh/plbook/book.pdf

Simovici, D. A. (1999). *Introduction to the Theory of Formal Languages*. Springer-Verlag.

Soto Romero, M. (2025a). *Alcance estático y dinámico* (Nota de Clase) (Lenguajes de Programación, Semestre 2026-1). Facultad de Ciencias, Universidad Nacional Autónoma de México.

Soto Romero, M. (2025b). *Ambientes de evaluación* (Nota de Clase) (Lenguajes de Programación, Semestre 2026-1). Facultad de Ciencias, Universidad Nacional Autónoma de México.

Soto Romero, M. (2025c). *El Cálculo Lambda como núcleo para funciones* (Nota de Clase) (Lenguajes de Programación, Semestre 2026-1). Facultad de Ciencias, Universidad Nacional Autónoma de México.

Soto Romero, M. (2025d). *Expresiones let* (Nota de Clase) (Lenguajes de Programación, Semestre 2026-1). Facultad de Ciencias, Universidad Nacional Autónoma de México.

Soto Romero, M. (2025e). *Máquinas abstractas* (Nota de Clase) (Lenguajes de Programación, Semestre 2026-1). Facultad de Ciencias, Universidad Nacional Autónoma de México.

Soto Romero, M. (2025f). *Resumen Proyecto 1 Lenguajes de Programación* (Nota de Clase) (Lenguajes de Programación, Semestre 2026-1). Facultad de Ciencias, Universidad Nacional Autónoma de México.

Soto Romero, M. (2025g). *Semántica Estática* (Nota de Clase) (Lenguajes de Programación, Semestre 2026-1). Facultad de Ciencias, Universidad Nacional Autónoma de México.

Soto Romero, M. (2025h). *Sintaxis Abstracta* (Nota de Clase) (Lenguajes de Programación, Semestre 2026-1). Facultad de Ciencias, Universidad Nacional Autónoma de México.

Soto Romero, M. (2025i). *Sintaxis Concreta* (Nota de Clase) (Lenguajes de Programación, Semestre 2026-1). Facultad de Ciencias, Universidad Nacional Autónoma de México.