Lenguajes de Programación

Parsing a language for arithmetical expressions

Federico Olmedo
Ismael Figueroa

Two levels of syntaxes

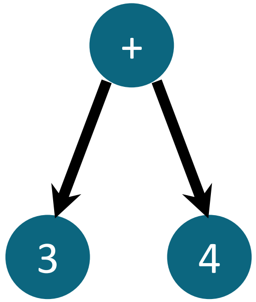
How we write our programs

3 + 4 — infix notation

(+ 3 4) — prefix notation

(3 4 +) — postfix notation

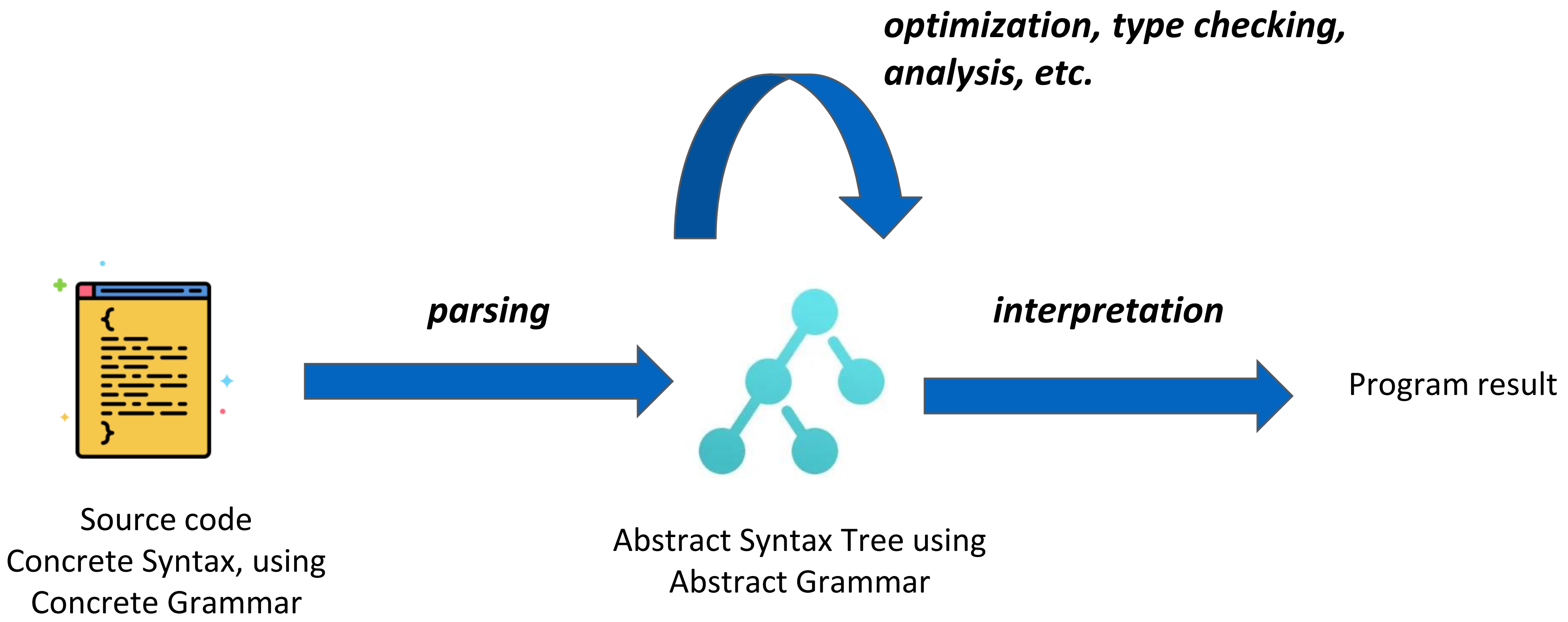
How we represent them
(to define its semantics)



"source" concrete syntax

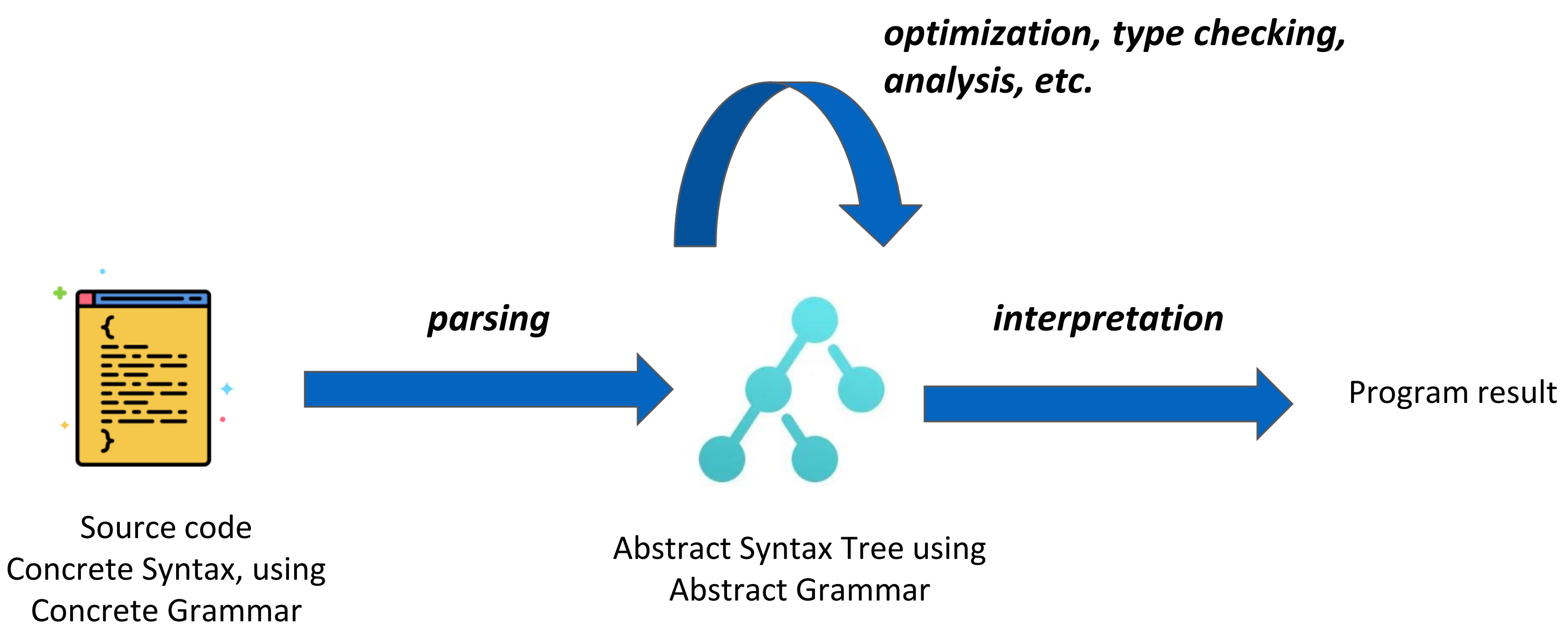
abstract syntax

Language processing pipeline



Abstract syntax enables us to focus on the implementation of semantics, regardless of the concrete syntax used by programmers

Implementing our language processing pipeline



- We will use inductive datatypes to describe the AST of our languages.
- We will use Racket's list-processing features to **parse** source code written as **s-expressions**

s-expressions

A *symbolic expression*, known as **s-expression**, is a notation invented in the Lisp language to represent nested tree-structured data. It is also used in Scheme and Racket as the base syntax.

It has **atoms,** which can be literal values and identifiers, and **nested expressions** which are recursively defined from other s-expressions.

```
#|
sexpr ::= atom
        | (sexpr sexpr*)
|#
```

```
1
x
(+ x y)
(define (f z) (+ z 1))
```

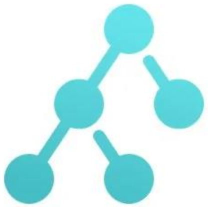
The source code for all our interpreters will be written using s-expressions.
In other words, we will use the same **prefix parenthetical syntax** used in Scheme or Racket.

Parsing a core language for arithmetic

We want to parse programs that perform additions on numbers. We need to define a concrete syntax/grammar and an abstract syntax/grammar.

Source code in “**input**” ***concrete grammar***, using s-expressions.

parsing



Abstract Syntax Tree, which follows the “output” abstract grammar.

```
#|
<s-expr> ::= <num>
           | (list '+ <s-expr> <s-expr>)
|#
```

```
#|
<expr> ::= (num <num>)
         | (add <expr> <expr>)
|#
```

The concrete and abstract grammars are not necessarily equal

Parsing a core language for arithmetic

Source code

```
#|
<expr> ::= (num <num>)
         | (add <expr> <expr>)
|#
```

parsing

Abstract
Syntax Trees

```
(deftype Expr
    (num n)
    (add l r))
```

We need to transform programs in source code into ASTs in the **`Expr`** datatype. For this, we need a **`parse`** **function**.

```
'1          (parse '1)         (num 1)
'14         (parse '14)        (num 14)
'{+ 1 2}    (parse '{+ 1 2})   (add (num 1) (num 2))
```

Source code s-exprs
written as Racket
quoted expressions

Abstract Syntax Trees as
`Expr` values in Racket

https://docs.racket-lang.org/guide/quote.html

Parsing a core language for arithmetic

Source code

```
#|
<expr> ::= (num <num>)
         | (add <expr> <expr>)
|#
```

parsing

Abstract
Syntax Trees

```
(deftype Expr
  (num n)
  (add l r))
```

```
;; parse :: s-expr -> Expr
;; Parses source code to Expr AST.
(define (parse s-expr)
  (match s-expr
    [n #:when (number? n) (num n)]
    [(list '+ l-expr r-expr)
     (add (parse l-expr) (parse r-expr))]))
```

The parser can also be mechanically derived from the BNF and/or the syntax of the s-expressions in the source language.

→ Extending the Language Syntax

How do we extend our language syntax?

Extending our language syntax involves several steps:

EXTEND CONCRETE SYNTAX: how will developers write programs using the new features?

EXTEND ABSTRACT SYNTAX: what information will we extract from the concrete syntax, for later interpretation of programs?

EXTEND PARSER: update the parsing function to correctly parse the new syntax and capture the abstract information

Adding a new arithmetic operation: subtraction

how will developers write programs using the new features?

EXTEND CONCRETE SYNTAX:

```
#|
<s-expr> ::= <num>
           | (list '+ <s-expr> <s-expr>)
           | (list '- <s-expr> <s-expr>)
|#
```

what information will we extract from the concrete syntax,
for later interpretation of programs?

EXTEND ABSTRACT SYNTAX:

```
#|
<expr> ::= (num <num>)
         | (add <expr> <expr>)
         | (sub <expr> <expr>)
|#
(deftype Expr
  (num n)
  (add l r)
  (sub l r))
```

Adding a new arithmetic operation: subtraction

update the parsing function to correctly parse the new syntax and capture the abstract information

EXTEND PARSER:

```
;; parse :: s-expr -> Expr
;; Parse an s-expr into an Expr.
(define (parse s-expr)
  (match s-exp
    [(? number? n) (num n)]
    [(list '+ l r) (add (parse l) (parse r))]
    [(list '- l r) (sub (parse l) (parse r))]))
```

Other arithmetic expressions are added in a similar way.

Note we used a different pattern to match on numbers.

- Expressions with conditionals

Extending the language with conditionals

We want to extend our language with a way to check whether a value is zero. More specifically, we will add the **`if0`** expression.

With this expression we will be able to write programs such as:

```
'{if0 {- 3 3} 1 2}

'{if0 {- 2 3} 1 2}
```

As well as nested expressions containing `if0`.

Adding `if0` conditional expression

EXTEND CONCRETE SYNTAX:

```
#|
<s-expr> :: = <num>
           | (list '+ <s-expr> <s-expr>)
           | (list '- <s-expr> <s-expr>)
           | (list 'if0 <s-expr> <s-expr> <s-expr>)
|#
```

EXTEND ABSTRACT SYNTAX:

```
#|
<expr> ::= (num <num>)
         | (add <expr> <expr>)
         | (sub <expr> <expr>)
         | (if0 <expr> <expr> <expr>)
|#
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f))
```

Adding `if0` conditional expression

EXTEND PARSER:

```
;; parse :: s-expr -> Expr
;; Parse an s-expr into an Expr.
(define (parse s-expr)
  (match s-exp
    [(? number? n) (num n)]
    [(list '+ l r) (add (parse l) (parse r))]
    [(list '- l r) (sub (parse l) (parse r))]
    [(list 'if0 c t f) (if0 (parse c) (parse t) (parse f))]))
```

Exercises

Boolean values

Extend the language syntaxes to support boolean values.
Use the same **`#t`** and **`#f`** literal values as in Racket

Boolean operations

Extend the language syntaxes to support boolean operations: **`not`**, **`and`**, **`or`**.

Numerical comparator

Extend the language syntaxes to support numerical comparators: **`<`**, **`<=`** , **`=`**, **`>`**, **`>=`**

Other arithmetical operations

Extend the language syntaxes to support other arithmetical operations, such as multiplication, logarithms, etc.

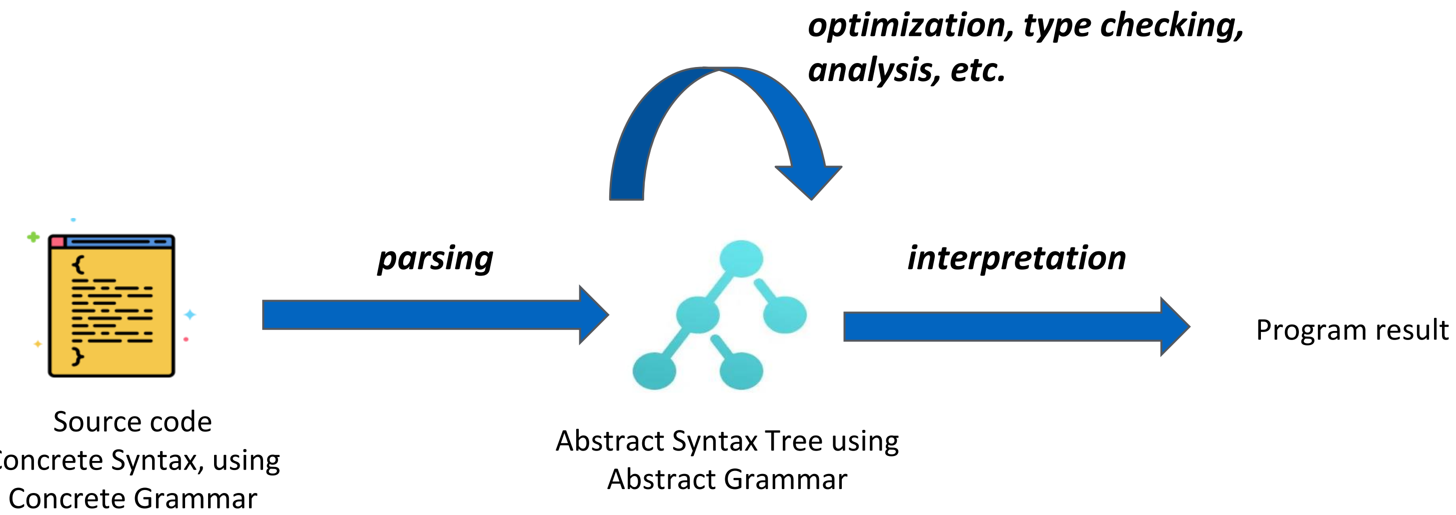
Arbitrary **`if`** expression

Extend language syntaxes to support arbitrary conditionals `(if c t f)`

Takeaways

On parsers

Program’s text must be translated into a more abstract representation for defining the program semantics. This is parsers’ job!!!



Abstract syntax opens the door to many techniques for program analysis, transformation, and optimization!

Lecture material

Bibliography

- Programming Languages: Application and Interpretation (1st Edition)
  Shriram Krishnamurthi [Download]
  Chapter 3

Source code

- Interpreter of arithmetical expressions [Download]
- Interpreter of arithmetical expressions with conditionals [Download]