Parsing II: Parsing in OCaml

CAS CS 320: Principles of Programming Languages

Thursday, March 21, 2024

Administrivia

- Homework 6 due Friday, Mar 22 (tomorrow), by 11:59 pm.
- Homework 7 posted Friday, Mar 22 (tomorrow), and due Friday, Mar 29.
- Withdraw deadline (grade W) is Friday, Mar 29.

REMINDER FROM PRECEDING LECTURE

Parsing I: An Introduction

- Get a sense of what parsing is, starting with lexical analysis.
- Look briefly at the general parsing problem.
- Look at recursive-decent as a first attempt at a simple parsing procedure.

REMINDER FROM PRECEDING LECTURE

Some important notions:

- parser generator
- lexical analysis
- lexeme
- token
- parsing
- recursive-decent

Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a ***sequence of characters*** into a ***sequence of tokens.***

Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a *sequence of characters* into a *sequence of tokens.*

think of a token as consisting of a ***token name*** and an *(optional)* ***token value***.

Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a ***sequence of characters*** into a ***sequence of tokens.*** **(following table is courtesy of Wikipedia)**

Examples of common tokens

Token name	Explanation	Sample token values
identifier	Names assigned by the programmer.	`x`, `color`, `UP`
keyword	Reserved words of the language.	`if`, `while`, `return`
separator/ punctuator	Punctuation characters and paired delimiters.	`}`, `(`, `;`
operator	Symbols that operate on arguments and produce results.	`+`, `<`, `=`
literal	Numeric, logical, textual, and reference literals.	`true`, `6.02e23`, `"music"`
comment	Line or block comments. Usually discarded.	`/* Retrieves user data */`, `// must be negative`
whitespace	Groups of non-printable characters. Usually discarded.	-

Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a *sequence of characters* into a *sequence of tokens.*

EXAMPLE: consider the following expression

```
x = a + b * 2 ;
```

a lexical analysis of this expression may produce the following sequence of 8 tokens:

```
[(identifier, x), (operator, =), (identifier, a), (operator, +),
(identifier, b), (operator, *), (literal, 2), (separator, ;)]
```

Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a *sequence of characters* into a *sequence of tokens.*

EXAMPLE: consider the following expression

```
x = a + b * 2 ;
```

a lexical analysis of this expression may produce the following sequence of 8 tokens:

```
[(identifier, x), (operator, =), (identifier, a), (operator, +),
(identifier, b), (operator, *), (literal, 2), (separator, ;)]
```

another compiler / interpreter may produce instead:

```
x (id) , = (binop) , a (id) , + (binop) , b (id) , * (binop) ,
2 (lit), ; (punctuation)
```

Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a *sequence of characters* into a *sequence of tokens*.

QUESTION: what are ***lexemes*** in relation to ***tokens***?

Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a *sequence of characters* into a *sequence of tokens*.

QUESTION: what are ***lexemes*** in relation to ***tokens***?

the demarcation between the two is not always stated in the same way in all books and articles . . .

but basically:

- a ***lexeme*** is a string of characters of a certain kind or type (e.g., a string literal, a sequence of chars).
- the ***lexeme***'s kind or type combined with its value is a ***token***, which can be given to a parser.

Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a *sequence of characters* into a *sequence of tokens.*

QUESTION: what is ***parsing*** in relation to ***tokens***?

Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a *sequence of characters* into a *sequence of tokens*.

QUESTION: what is ***parsing*** in relation to ***tokens***?

basically:

- the ***parser*** takes the ***sequence of tokens*** and checks them against the grammar of the language.
- this grammar defines how ***tokens*** can be combined to form ***valid statements*** and expressions.
- if the ***tokens*** follow the grammar rules, the parser constructs an ***abstract syntax tree*** (or also a ***parse tree***), representing the structure of the source code.

Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a *sequence of characters* into a *sequence of tokens.*

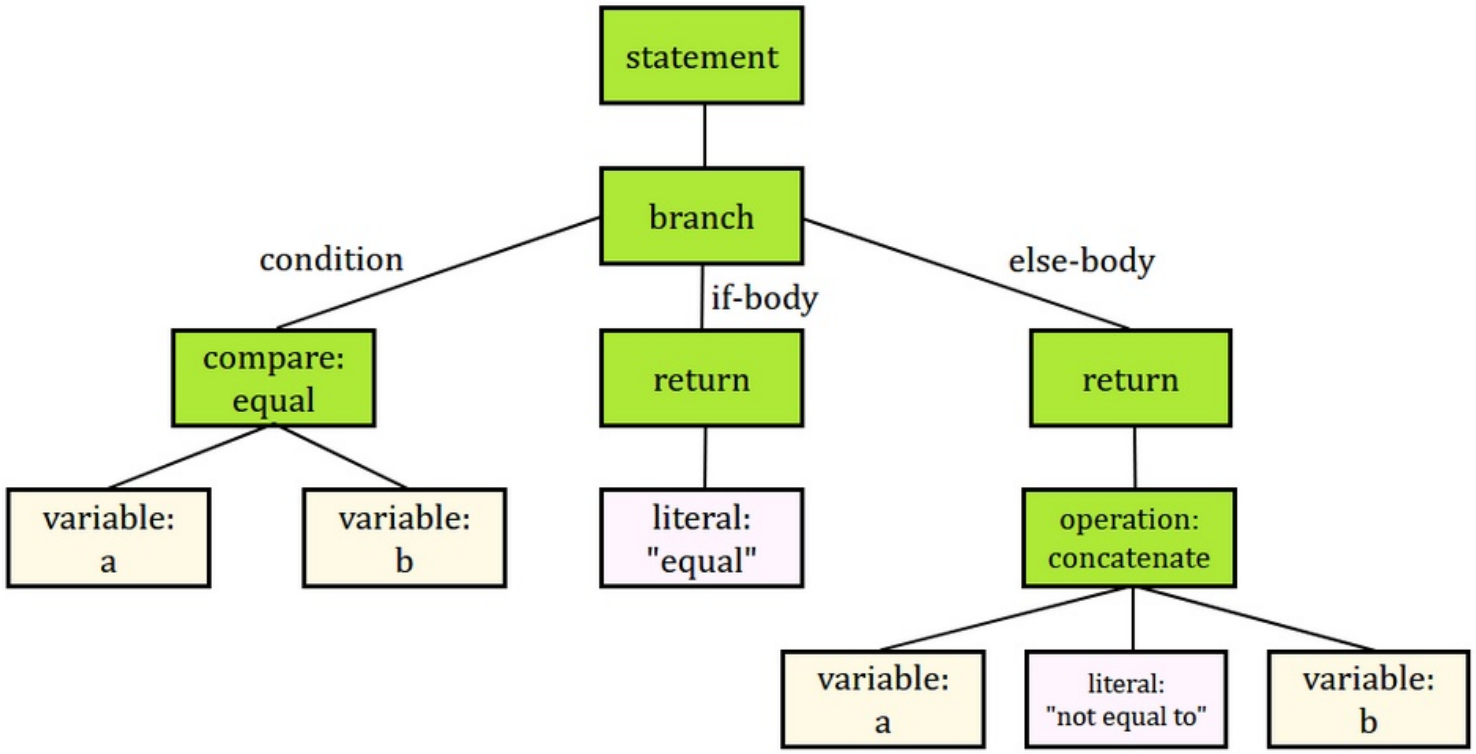
EXAMPLE: consider the following piece of code:

```
if a = b
  then
    return "equal"
  else
    return a + " not equal to " + b
```

the parser may produce an AST of the form shown below:

Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a ***sequence of characters*** into a ***sequence of tokens*** – translated into an AST by the parser:



Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a *sequence of characters* into a *sequence of tokens*.

REMARK:

- there are many ways in which the parser can return an AST from the sequence of tokens,
- and the form of an AST will vary based on the language and tool used to create it,
- however, an AST should have the property of completely representing the source code in a reproducible manner.

Elaborating a little more:

lexical analysis or ***lexical tokenization*** is the process of converting a *sequence of characters* into a *sequence of tokens*.

REMARK:

- there are many ways in which the parser can return an AST from the sequence of tokens,
- and the form of an AST will vary based on the language and tool used to create it,
- however, an AST should have the property of completely representing the source code in a reproducible manner.

the AST is then used in the next stage of interpretation/ compilation process, such as *semantic analysis* - and, in the case of compilation, *code generation*.

INTERLUDE (about formal grammars & languages)

INTERLUDE (about formal grammars & languages)

- we covered a tiny fraction of formal-language theory - regular grammars and context-free grammars (aka BNF).

INTERLUDE (about formal grammars & languages)

- we covered a tiny fraction of formal-language theory - regular grammars and context-free grammars (aka BNF).
- and that tiny fraction did not touch all aspects related to programming and programming languages.

INTERLUDE (about formal grammars & languages)

- we covered a tiny fraction of formal-language theory - regular grammars and context-free grammars (aka BNF).
- and that tiny fraction did not touch all aspects related to programming and programming languages.
- for example, if there are context-free grammars, what is *context-sensitive*? if there are right-linear grammars, what is *linear* and what is *left-linear*? etc.

INTERLUDE (about formal grammars & languages)

- we covered a tiny fraction of formal-language theory - regular grammars and context-free grammars (aka BNF).
- and that tiny fraction did not touch all aspects related to programming and programming languages.
- for example, if there are context-free grammars, what is *context-sensitive*? if there are right-linear grammars, what is *linear* and what is *left-linear*? etc.
- and within that tiny fraction, many of our claims were approximate or stated without proof (e.g. well-formed sequences of parentheses cannot be generated by a regular grammar).

INTERLUDE (about formal grammars & languages)

here something in that tiny fraction related to parsing:

a *normal form* is a standard way of viewing a mathematical or formal object, e.g. the fraction 1/2 is a *normal form* for 5/10 and 3/6.

INTERLUDE (about formal grammars & languages)

here something in that tiny fraction related to parsing:

a *normal form* is a standard way of viewing a mathematical or formal object, e.g. the fraction 1/2 is a *normal form* for 5/10 and 3/6.

example in language theory: a context-free grammar is in *Chomsky normal form* if each production is of one of the following forms:

(1) $S ::= \varepsilon$

(2) $A ::= B\ C$ A, B, C arbitrary non-terminals

(3) $A ::= a$ A, a arbitrary non-terminal and terminal

if (1) is one of the production rules, then neither B nor C is S in clause (2).

INTERLUDE (about formal grammars & languages)

here something in that tiny fraction related to parsing:

a *normal form* is a standard way of viewing a mathematical or formal object, e.g. the fraction 1/2 is a *normal form* for 5/10 and 3/6.

example in language theory: a context-free grammar is in *Chomsky normal form* if each production is of one of the following forms:

(1) $S ::= \varepsilon$

(2) $A ::= B\ C$ A, B, C arbitrary non-terminals

(3) $A ::= a$ A, a arbitrary non-terminal and terminal

if (1) is one of the production rules, then neither B nor C is S in clause (2). FACT: Every context-free grammar can be transformed into Chomsky normal form.

(THIS PAGE INTENTIONALLY LEFT BLANK)