Compilers

INF-400

Burak Arslan
ext-inf400@burakarslan.com

Galatasaray Üniversitesi

Lecture V
2023-11-02

Course website

burakarslan.com/inf400

Homework II

Homework II will be out today!

Deadline: **2023-11-08 23:59**

Homework I

Any questions?

Parsing

Can we keep on using regular expressions?

Are regular expressions enough to specify a complete programming language?

Parsing

Can we keep on using regular expressions?

What is the regular expression for the language of balanced parentheses?

$$L = \{(^i)^i \mid i \geq 0\}$$

Parsing

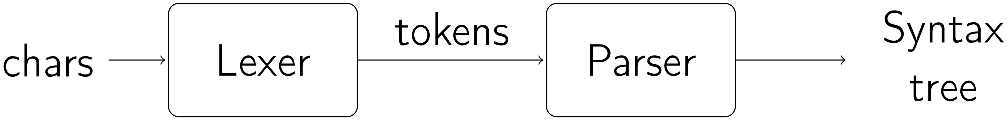
Can we keep on using regular expressions?

Finite Automata can't:

- Count the number of times a state was visited.
- As a consequence, they can't represent **nested structures**.

Parsing

Purpose of parsing



Parsing

Purpose of parsing

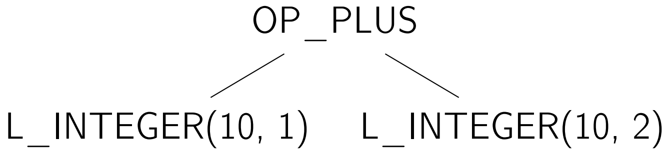
We generate an **Abstract Syntax Tree** – with as little information as possible.

- Parse Tree / Concrete syntax tree: contains every detail. Language-dependent structure
- Abstract Syntax Tree: Just enough to do semantic analysis. Mostly language-independent

Parsing

Purpose of parsing

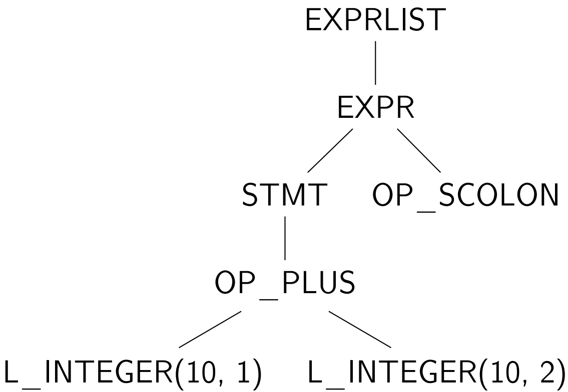
Let's see the abstract syntax tree for: `1 + 2;`



Parsing

Purpose of parsing

Let's see the parse tree for: `1 + 2;`



Parsing

Purpose of parsing

Parsers also apply further validation:

Not all token arrays are valid programs!

Statements in programming languages are **nested structures**.

Context Free Grammars.

are a perfect match.

Context Free Grammars: Definition:

- A set of terminals T
- A set of nonterminals N
- A start symbol s
- A set of productions

Parsing

Context Free Grammars

We will use a tool named **Bison**: Conventions:

- Terminal names are uppercase.
- Non-terminals are all lowercase.
- Starting symbol is the first symbol.

Parsing

Context Free Grammars

Bison algorithm:

- Start from the starting symbol given input token array.
- Replace non terminals by one of the productions on the right [1]
- Repeat until only terminals remain

[1] This point is doint a lot of work here ☺

Parsing

Context Free Grammars

Bison algorithm:

- Start from the starting symbol given input token array.
- Replace non terminals by one of the productions on the right [1]
- Repeat until only terminals remain

In other words:

- $s \rightarrow T_0 \ldots T_n$

[1] This point is doint a lot of work here ☺

Parsing

Context Free Grammars

Terminals;

- are supposed to be the tokens recognized by the lexer
- can't be replaced once generated, hence the name "terminal"

Parsing

Context Free Grammars

Let's see the grammar for the following language: $L = \{(^i)^i, i \geq 0\}$

$$\text{expr} \rightarrow (\text{expr})|\varepsilon$$

Or, in bison notation:

```
expr
  : %empty
  | '(' expr ')'
  ;
```

Parsing

Context Free Grammars

Simple arithmetic example:

$$e \rightarrow e + e | e \times e | (e) | \texttt{ID}$$ [2]

Some strings from the above language:

```
 x
 x + (y)
 x + y * z
(x + y) * z
```

[2] Remember the IDENTIFIER from lexer project? It's the same thing

Parsing

Context Free Grammars

Derivation: A sequence of productions leading to a string of terminals.

Given:

$$e \rightarrow e + e | e \times e | (e) | \mathtt{ID}$$

Let's see the derivation of:

`a * b + c`

- e

Parsing

Context Free Grammars

Derivation: A sequence of productions leading to a string of terminals.

Given:

$$e \rightarrow e + e | e \times e | (e) | \mathtt{ID}$$

Let's see the derivation of:

```
a * b + c
```

- e
- $e + e$

Parsing

Context Free Grammars

Derivation: A sequence of productions leading to a string of terminals.

Given:

$$e \to e + e | e \times e | (e) | \texttt{ID}$$

Let's see the derivation of:

```
a * b + c
```

- e
- $e + e$
- $e \times e + e$

Parsing

Context Free Grammars

Derivation: A sequence of productions leading to a string of terminals.

Given:

$$e \rightarrow e + e | e \times e | (e) | \mathtt{ID}$$

Let's see the derivation of:

```
a * b + c
```

- e
- $e + e$
- $e \times e + e$
- $ID \times e + e$

Parsing

Context Free Grammars

Derivation: A sequence of productions leading to a string of terminals.

Given:

$$e \rightarrow e + e | e \times e | (e) | \texttt{ID}$$

Let's see the derivation of:

```
a * b + c
```

- e
- $e + e$
- $e \times e + e$
- $ID \times e + e$
- $ID \times ID + e$

Parsing

Context Free Grammars

Derivation: A sequence of productions leading to a string of terminals.

Given:

$$e \rightarrow e + e | e \times e | (e) | \texttt{ID}$$

Let's see the derivation of:

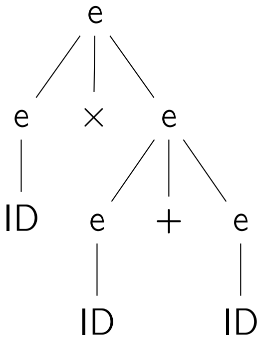
```
a * b + c
```

- e
- $e + e$
- $e \times e + e$
- $ID \times e + e$
- $ID \times ID + e$
- $ID \times ID + ID$

Parsing

Context Free Grammars

Derivation of `a * b + c`:

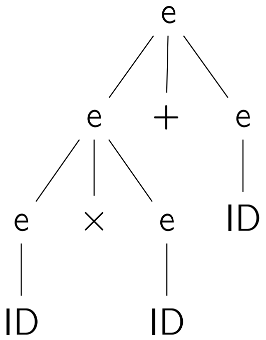


- e
- $e \times e$
- $e \times e + e$
- $ID \times e + e$
- $ID \times ID + e$
- $ID \times ID + ID$

Parsing

Context Free Grammars

Derivation of `a * b + c`:



- e
- $e \times e$
- $e \times e + e$
- $e \times e + ID$
- $e \times ID \times ID$
- $ID \times ID \times ID$

Ambiguity is **BAD**

Ambiguity means your language contains **ill-defined** code fragments[3].

[3] ie. Code fragments with more than one meaning

Parsing

Context Free Grammars

Dealing with ambiguity:

- Add visible precedence markers (tokens)
- Add implicit precedence markers (`%left` and `%right`)
- Write a non-ambiguous grammar

Parsing

Context Free Grammars

Dealing with ambiguity:

- Add visible precedence markers (tokens, eg. parentheses)
- Add implicit precedence markers (`%left` and `%right`)
- Write a non-ambiguous grammar

Parsing

Context Free Grammars

A non-ambiguous version of the below grammar

$$e \rightarrow e + e | e \times e | \mathtt{ID}$$

could be as follows:

$$\begin{aligned} e &\rightarrow e + e | f + e | f \\ f &\rightarrow f \times e | f \times f | \mathtt{ID} \end{aligned}$$

... where the multiplication has precedence over addition

Next Up

- **Shift-reduce** or **Bottom-up** parsers (what bison does)
- (Maybe) other parsing algorithms
- The rest of the kiraz grammar