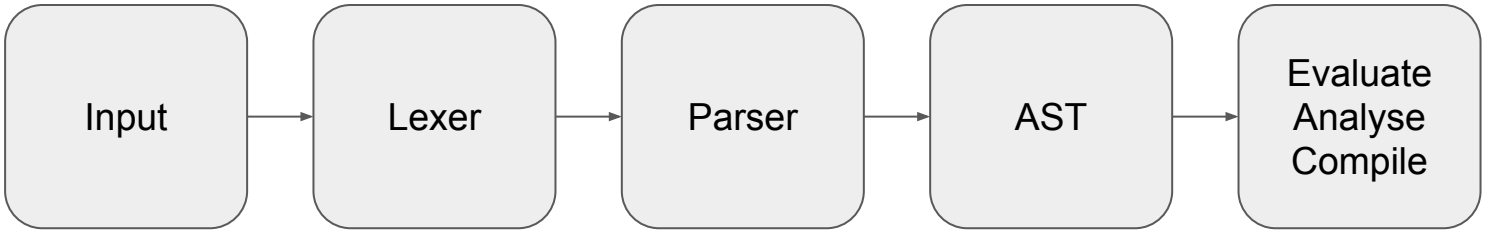
Parsowanie języczków

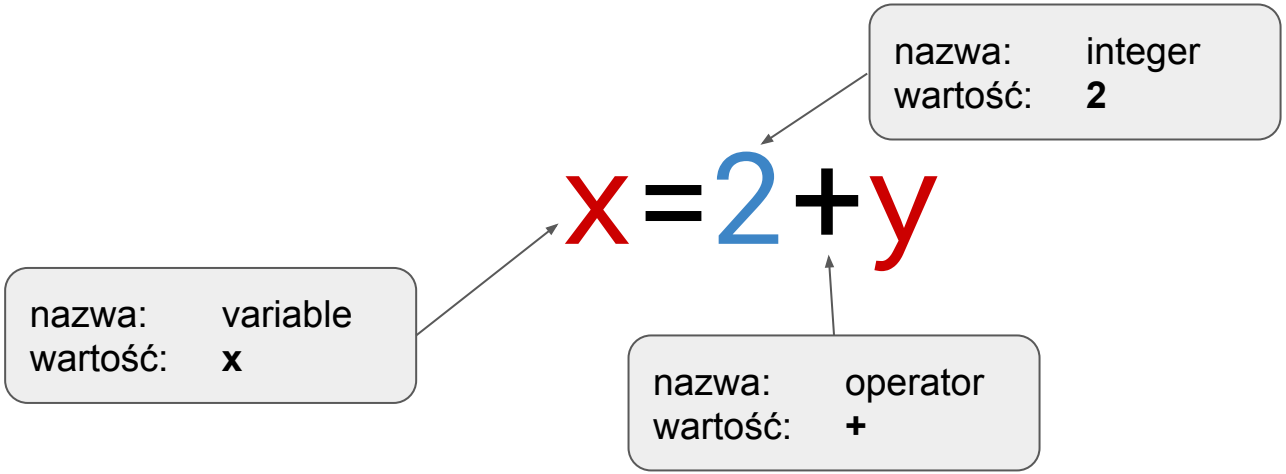
Gramatyki, tokeny, DSLki i inne takie

Plan



Lexer & Token

- Analizują wejście wyszukując “słów” zwanymi tokenami
- Tokeny zawierają nazwę (typ) oraz ewentualnie wartość
- Tokeny opisujemy wyrażeniami regularnymi



Przykłady tokenów

- identifier `x` `age` `counter`
- keyword `if` `while` `for`
- operator `+` `-` `*`
- literal `"str"` `32` `0xff`

Wyrażenia regularne

- Jak nazwa wskazuje, wykrywa tekst który się “powtarza” według regularnego wzoru
- opisuje się osobnym językiem który wygląda jak efekt kota na klawiaturze
- wyrażenia regularne pozwalają wyciągnąć tylko tę część która nas interesuje

Wyrażenia regularne - przykłady

Wyrażenie	Przykłady
a+	a aa aaaaaaaaaa
ab*	a ab abbbbbbb
[a-z_]+	marek ziemniak_z_cebulka
[0-9]+	0 123 20938492849
monsters?	monster monsters
foo..	fooab foocc foo13 foo&%
wom(bat|en)	wombat women
[0-9]+(\.[0-9]+)?	000 123 0.3223 0.2

Tokenowy przykład!

Boże marek pokaż ten kod w końcu

Gramatyka

- Jak już mamy tokeny, to możemy stworzyć gramatykę
- Gramatyka już nie analizuje tokenów, tylko operuje tokenami (analogicznie jak dzięki słowom możemy zbadać gramatykę zdania)
- Gramatyka składa się z:
  - zasad (symbol nieterminalny) - składa się z innych zasad oraz tokenów
  - tokenów (symboli terminalnych) - te zawierają nazwę oraz wartość
  - symbolu startowego
- Gramatyka sprawdza poprawność podanego wyrażenia
- Dzięki gramatyce możemy stworzyć AST - Abstract Syntax Tree

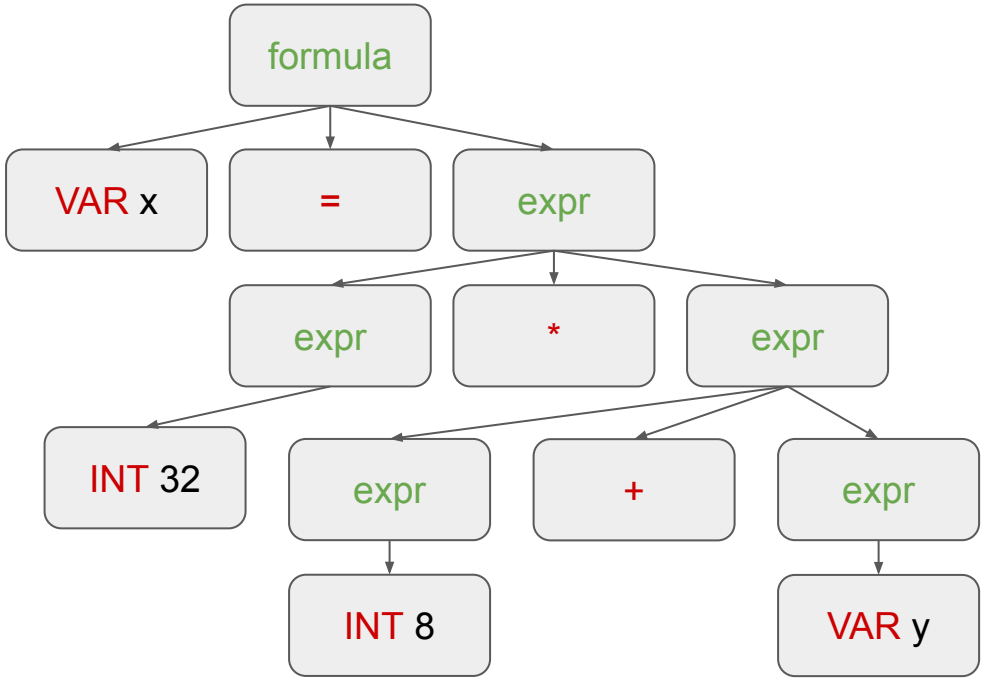
Przykład Gramatyki

- start: `formula`
- tokeny: { `VAR + * = INT` }
- zasady:
  - `formula` ⇒ `VAR = expr`
  - `expr` ⇒ `expr + expr`
  - `expr` ⇒ `expr * expr`
  - `expr` ⇒ `VAR`
  - `expr` ⇒ `INT`

Przykład Gramatyki

- formula ⇒ VAR = expr
- expr ⇒ expr + expr
- expr ⇒ expr * expr
- expr ⇒ VAR
- expr ⇒ INT

```
x = 32 * 8 + y
```



Kolejność operatorów

- najfaniej jest gdy parser to już obsługuje
- https://en.wikipedia.org/wiki/Shunting-yard_algorithm
- Można też ustalić to gramatyką:
  - `formula  ⇒ VAR = add_expr`
  - `add_expr ⇒ add_expr + add_expr`
  - `add_expr ⇒ expr`
  - `expr     ⇒ expr * expr`
  - `expr     ⇒ VAR`
  - `expr     ⇒ INT`

Parsery

- LL - Left to Left parser
  - pobiera z lewej strony token oraz analizuje z lewej strony zasady
  - Są banalne w napisaniu
  - Są szybkie
  - Wymagają modyfikacji gramatyki, inaczej dojdzie do błędu
  - Python do wersji 3.8 (czyli ok. 25 lat!) stosował z LL parser
- LR - Left to Right parser
  - pobiera z lewej strony token oraz analizuje z prawej strony zasady
  - Potrafi obsłużyć o wiele bardziej zaawansowane gramatyki
  - wyróżniamy wiele rodzin jak LR(0) LR(1) SLR(1) czy LALR(1)
  - Algorytm LR jest tak zasobożerny, że po powstaniu nie było komputerów które mogłyby przeanalizować gramatykę tymże algorytmem

Parser LL - pułapki

Gramatyka:

- `expr ⇒ expr + expr`
- `expr ⇒ INT`
- `expr ⇒ VAR`

będzie rozwijana tak:

`expr -> expr + expr -> expr + expr + expr -> expr + … + ∞`

(w skrócie, zapętli się)

Parser LL - pułapki

Poprawna gramatyka:

- `expr  ⇒ value + expr`
- `value ⇒ INT`
- `value ⇒ VAR`

wynik:

```
expr -> value + expr -> INT + value + expr -> …
```

Parser LL - pułapki

Gramatyka:

- `expr` ⇒ `value + expr`
- `expr` ⇒ `value * expr`
- `expr` ⇒ `value`

Ze względu, że parser LL patrzy tylko na lewą stronę zasady, nie będzie wiedział, której zasady może użyć (użyje pierwszej lub zgłosi błąd)

Parser LL - pułapki

Poprawna gramatyka:

- `expr ⇒ value op_expr`
- `op_expr ⇒ + expr`
- `op_expr ⇒ * expr`
- `op_expr ⇒ ε`

wynik:

```
expr -> value op_expr -> value * expr

value * expr -> value * value op_expr -> value * value
```

Analiza AST

Jak już posiadamy AST (Abstract Syntax Tree) to możemy:

- Zmienić go na bytecode (wirtualne maszyny np. Erlangowa czy Java)
- Zmienić go na assembler (w tej chwili lepiej stworzyć LLVM)
- Wykonać dynamicznie po analizie AST (Python, Ruby, JS)

Co dalej?

- Parserów jest DUŻO
- https://github.com/dabeaz/sly
- Yacc & Flex (C/C++)
- Można zacząć od swoich DSL (Domain Small Language)
- Tworzenie swojego języka programowania to ciężki temat:
  - zarządzanie pamięcią (w tym jej czyszczenie)
  - scope zmiennych w blokach kodu
  - funkcje, klasy, pętle, fore, include
- Jak już chcesz ten język robić, to zacznij od LISP-like
- Parsowanie DSL / języka to bardzo duży level up w programowaniu
- https://github.com/firemark/notebooks/tree/master/ll-parser

Q&A