Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Informatyki

Projekt dyplomowy

Implementacja narzędzi lex i yacc z wykorzystaniem metaprogramowania

Implementation of lexical analyzer (lex) and parser generator (yacc) tools using metaprogramming techniques

Autorzy: Bartosz Buczek, Bartłomiej Kozak
Kierunek studiów: Informatyka
Opiekun pracy: dr inż. Tomasz Służalec

Kraków, 2025

Spis treści

Rozdział 1

Cel prac i wizja projektu

1.1. Charakterystyka problemu

Leksery i parsery są kluczowymi elementami w procesie tworzenia interpreterów i kompilatorów języków programowania. Pozwalają one przekształcić kod źródłowy napisany przez programistę na reprezentację wewnętrzną, wykorzystywaną później przez dalsze etapy przetwarzania kodu.

Analiza leksykalna wykonywana przez lekser polega na rozdzieleniu kodu źródłowego na jednostki logiczne, zwane leksemami. Parser natomiast wykonuje analizę składniową w celu ustalenia struktury gramatycznej tekstu i jej zgodności z gramatyką języka.

Celem pracy inżynierskiej jest stworzenie narzędzia *ALPACA* (Another Lexer Parser And Compiler Alpaca) w języku Scala, które implementuje funkcjonalności powszechnie stosowane w budowie lekserów i parserów.

1.2. Motywacja projektu

Projekt ma na celu stworzenie nowoczesnego narzędzia do generowania lekserów i parserów w języku Scala, łączącego zalety istniejących rozwiązań z nowoczesnym podejściem technologicznym. Jego główne cele to:

1. Stworzenie intuicyjnego API.

2. Opracowanie obszernej dokumentacji.

3. Rozbudowana diagnostyka błędów.

4. Poprawa wydajności względem rozwiązań w języku Python.

5. Integracja z popularnymi środowiskami programistycznymi (IDE).

Proponowane rozwiązanie łączy nowoczesne podejście technologiczne z praktycznym zastosowaniem w edukacji i programowaniu. Może on służyć jako narzędzie dydaktyczne, ułatwiając naukę teorii kompilacji, w pracach badawczych, a także jako kompleksowe narzędzie do tworzenia praktycznych rozwiązań.

1.3. Przegląd istniejących rozwiązań

Dostępne na rynku rozwiązania umożliwiają tworzenie analizatorów, jednak charakteryzują się ograniczeniami związanymi z wydajnością, wysokim progiem wejścia i diagnostyką błędów.

1.3.1. Lex, Yacc

Lex[1] i *Yacc*[2] to klasyczne, dobrze ugruntowane narzędzia, które odegrały kluczową rolę w tworzeniu setek współczesnych języków programowania. Definicja leksera i parsera w tych systemach odbywa się poprzez specjalnie zaprojektowaną składnię konfiguracyjną. Mimo pewnych zalet, jego złożoność i wysoki próg wejścia mogą stanowić wyzwanie.

Ponieważ *Lex* i *Yacc* zostały zaprojektowane do współpracy z językiem C, ich integracja z nowoczesnymi językami programowania bywa utrudniona. Rozszerzanie tych narzędzi o dodatkowe, specyficzne funkcjonalności jest skomplikowane, co ogranicza ich elastyczność. Brak wsparcia dla współczesnych środowisk programistycznych (IDE) dodatkowo obniża komfort użytkowania w porównaniu z nowoczesnymi alternatywami.

```
{
/*%%%*/
value_expr($3);
$1->nd_value = $3;
$$ = $1;
/*%
$$ = dispatch2(massign, $1, $3);
%*/
}
| var_lhs tOP_ASGN command_call
{
value_expr($3);
$$ = new_op_assign($1, $2, $3);
}
| primary_value '[' opt_call_args rbracket tOP_ASGN command_call
{
/*%%%*/
NODE *args;

value_expr($6);
if (!$3) $3 = NEW_ZARRAY();
args = arg_concat($3, $6);
if ($5 == tOROP) {
    $5 = 0;
}
else if ($5 == tANDOP) {
    $5 = 1;
}
$$ = NEW_OP_ASGN1($1, $5, args);
fixpos($$, $1);
/*%
$$ = dispatch2(aref_field, $1, escape_Qundef($3));
$$ = dispatch3(opassign, $$, $5, $6);
%*/
}
```

Listing 1.1: Fragment definicji parsera Ruby w technologii Yacc

1.3.2. PLY, SLY

PLY[3] i jego nowszy odpowiednik *SLY*[4] to biblioteki inspirowane narzędziami Lex i Yacc. Oferują elastyczne podejście do budowy parserów, umożliwiając samodzielną implementację obsługi leksemów, budowę drzewa AST, czy dodatkowe funkcjonalności takie jak obliczanie numeru linii w lekserze.

Głównym ograniczeniem PLY i SLY jest implementacja w języku Python. Ze względu na interpretowany charakter oraz dynamiczne typowanie, parsery te charakteryzują się niską wydajnością, a brak statycznego typowania utrudnia wykrywanie błędów na etapie kompilacji. Przy implementacji parserów z użyciem biblioteki SLY w środowisku PyCharm obserwuje się wiele ostrzeżeń dotyczących potencjalnych naruszeń reguł, co często wymaga zastosowania mechanizmów supresji, aby uniknąć fałszywie pozytywnych wyników analizy statycznej kodu. Ponadto należy zaznaczyć, iż autor projektu informuje o braku dalszego rozwoju tych narzędzi[5].

Przykład 1.2 ilustruje kilka nieintuicyjnych, automatycznych mechanizmów obecnych w bibliotece *SLY*.

- Operator `@_()` jest zdefiniowany, aby automatycznie analizować tekst przy pomocy wyrażeń regularnych. Literały muszą być zawarte w cudzysłowie, a „zmienna” odpowiada za matchowany „typ”.
- Nazwa metody oznacza „typ” zwracany przez daną produkcję, czyli dla definicji `IF` należy najpierw odszukać wszystkie metody, które mają nazwę `condition`, gdyż są to możliwe produkcje.
- W krotce (sic!) `precedence` definiujemy pierwszeństwo operatorów, jednakże dodanie `% prec` pozwala nadpisać priorytet dla konkretnej reguły składniowej.
- Argument `p` pozwala na dostęp do kontekstu produkcji (np. numeru linii), ale także do zmiennych w patternu match w adnotacji. Jeśli zdefiniowany jest więcej niż jeden, to dodajemy numer do accesora, np. `expr1` jest odwołaniem się do drugiego wyrażenie `expr`. Jednocześnie, można to zrobić także poprzez odwołanie się do konkretnego indeksu obiektu `p`.

```
class MatrixParser(Parser):
tokens = MatrixScanner.tokens

precedence = (
    ('nonassoc', 'IFX'),
    ('nonassoc', 'ELSE'),
    ('nonassoc', 'EQUAL'),
)

@_('"{" instructions "}"')
def block(self, p: YaccProduction):
    raise NotImplementedError

@_('instruction')
def block(self, p: YaccProduction):
    raise NotImplementedError

@_('IF "(" condition ")" block %prec IFX')
def instruction(self, p: YaccProduction):
```

```
20      raise NotImplementedError
21
22  @_('IF "(" condition ")" block ELSE block')
23  def instruction(self, p: YaccProduction):
24      raise NotImplementedError
25
26  @_('expr EQUAL expr')
27  def condition(self, p: YaccProduction):
28      args = [p.expr0, p.expr1]
29      raise NotImplementedError
```

Listing 1.2: Fragment definicji parsera w Pythonie, wykorzystujacy bibliotekę SLY

Komunikaty błędów w bibliotece *SLY* są bardzo ograniczone, co obrazuje przykład 1.3, który po uruchomieniu informuje użytkownika błędem z fragmentu kodu **??**. Okazuje się, że problemem był brak atrybutu `ignore_comment` w definicji `Lexer`.

```
1  tokens = Scanner().tokenize("a = 1 + 2")
2  for tok in tokens:
3      print(tok)
```

Listing 1.3: Fragment niedziałajacego kodu w Pythonie, wykorzystujacy bibliotekę SLY

```
1    File "main.py", line 2, in <module>
2      for tok in tokens:
3                 ^^^^^^
4    File "Python\site-packages\sly\lex.py", line 374, in tokenize
5      _set_state(type(self))
6      ~~~~~~~~~~^^^^^^^^^^^^
7    File "Python\site-packages\sly\lex.py", line 367, in _set_state
8      _master_re = cls._master_re
9                   ^^^^^^^^^^^^^^
10 AttributeError: type object 'Scanner' has no attribute '_master_re'
```

Listing 1.4: Przykładowy komunikat błędu w bibliotece *SLY*

1.3.3. ANTLR

ANTLR[6] to kolejne rozwiązanie inspirowane narzędziami *Lex* i *Yacc*, oferujące zaawansowane mechanizmy analizy składniowej. Jego twórcy opracowali dedykowany język DSL, znany jako Grammar v4, który umożliwia definiowanie składni analizowanego języka. Na podstawie tej definicji *ANTLR* generuje parser w wybranym przez użytkownika języku programowania, takim jak Python, Java, C++ lub JavaScript.

Wspomaganie pracy z *ANTLR* w znacznym stopniu ułatwiają dedykowane wtyczki do środowisk Visual Studio Code oraz IntelliJ IDEA. Oferują one funkcjonalności, takie jak kolorowanie składni, autouzupełnianie kodu, nawigację do definicji leksemów oraz walidację błędów, co znacząco przyspiesza proces tworzenia parserów.

Jedną z kluczowych różnic *ANTLR* w porównaniu do innych narzędzi jest wykorzystanie gramatyki LL(*), podczas gdy klasyczne rozwiązania, takie jak Yacc czy SLY, implementują LALR(1). LL(*) jest bardziej intuicyjna i czytelna dla programistów, co ułatwia definiowanie reguł składniowych. Jednakże, jej zastosowanie wiąże się z większym zużyciem pamięci oraz niższą wydajnością w porównaniu do LALR(1).

Dodatkowym wyzwaniem podczas korzystania z *ANTLR* jest konieczność nauki składni DSL Grammar v4 oraz ograniczenie wsparcia dla narzędzi deweloperskich. Pełne wykorzystanie możliwości *ANTLR* wymaga korzystania z jednego z dedykowanych środowisk, co może stanowić istotne ograniczenie dla użytkowników preferujących inne IDE.

1.3.4. Scala parser combinators

Biblioteka *Scala parser combinators*[7] była popularnym sposobem na tworzenie parserów, lecz jak wynika z dokumentacji, „Trudno jest jednak zrozumieć ich działanie i jak zacząć. Po skompilowaniu i uruchomieniu kilku pierwszych przykładów, mechanizm działania staje się bardziej zrozumiały, ale do tego czasu może to być zniechęcające, a standardowa dokumentacja nie jest zbyt pomocna”[8].

1.3.5. ScalaBison

Z podsumowania artykułu na temat *ScalaBison*[9] wiadomo, że to praktyczny generator parserów dla języka Scala oparty na technologii rekurencyjnego wstępowania i zstępowania, który akceptuje pliki wejściowe w formacie *bison*. Parsery generowane przez *ScalaBison* używają bardziej informacyjnych komunikatów o błędach niż te generowane przez pierwowzór *bison*, a także szybkość parsowania i wykorzystanie miejsca są znacznie lepsze niż *scala-combinators*, ale są nieco wolniejsze niż najszybsze generatory parserów oparte na JVM.

Dodatkowo należy zaznaczyć, iż jest to rozwiązanie już niewspierane i stworzone w celach akademickich. Korzysta z przestarzałej wersji Scali, nie posiada wyczerpującej dokumentacji i liczba funkcjonalności jest bardzo ograniczona w porównaniu do np. technologii *SLY*.

1.3.6. parboiled2

parboiled2[10] to biblioteka w Scali umożliwiająca lekkie i szybkie parsowanie dowolnego tekstu wejściowego. Implementuje ona oparty na makrach generator parsera dla gramatyk wyrażeń parsujących (PEG), który działa w czasie kompilacji i tłumaczy definicję reguły gramatycznej na odpowiadający jej bytecode JVM. Niestety próg wejścia ze względu na skomplikowany i nieintuicyjny DSL jest wysoki. Zgodnie z przykładem 1.5, raportowanie błędów jest bardzo ograniczone (problem z implementacją wynika jedynie z różnic w liczbie parametrów funkcji).

```
1 [error] /Users/haoyi/Dropbox (Personal)/Workspace/scala-js-book/scalatexApi
     /src/main/scala/scalatex/stages/Parser.scala:60: overloaded
2 method value apply with alternatives:
3 [error] [I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, RR](f: (I, J
     , K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, scalatex.stages.Ast.
     Block.
4 Text, scalatex.stages.Ast.Chain, Int, scalatex.stages.Ast.Block) => RR)(
     implicit j: org.parboiled2.support.ActionOps.SJoin[shapeless.::[I,
5 shapeless.::[J,shapeless.::[K,shapeless.::[L,shapeless.::[M,shapeless.::[N,
     shapeless.::[O,shapeless.::[P,shapeless.::[Q,shapeless.::[R,
6 shapeless.::[S,shapeless.::[T,shapeless.::[U,shapeless.::[V,shapeless.::[W,
     shapeless.::[X,shapeless.::[Y,shapeless.::[Z,shapeless.
```

```
7 HNil]]]]]]]]]]]]]]]]]],shapeless.HNil,RR], implicit c: org.parboiled2.
    support.FCapture[(I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z,
    scalatex.
8 stages.Ast.Block.Text, scalatex.stages.Ast.Chain, Int, scalatex.stages.Ast.
    Block) => RR])org.parboiled2.Rule[j.In,j.Out] <and>
9 [error] [J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, RR](f: (J, K, L
    , M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, scalatex.stages.Ast.Block.
    Text,
10 scalatex.stages.Ast.Chain, Int, scalatex.stages.Ast.Block) => RR)(implicit
    j: org.parboiled2.support.ActionOps.SJoin[shapeless.::[J,
11 shapeless.::[K,shapeless.::[L,shapeless.::[M,shapeless.::[N,shapeless.::[O,
    shapeless.::[P,shapeless.::[Q,shapeless.::[R,shapeless.::[S,
12 shapeless.::[T,shapeless.::[U,shapeless.::[V,shapeless.::[W,shapeless.::[X,
    shapeless.::[Y,shapeless.::[Z,shapeless.HNil]]]]]]]]]]]]]]]],shapeless.
    HNil,RR], implicit c: org.parboiled2.support.FCapture[(J, K, L, M, N, O,
     P, Q, R, S,
13 T, U, V, W, X, Y, Z, scalatex.stages.Ast.Block.Text, scalatex.stages.Ast.
    Chain, Int, scalatex.stages.Ast.Block) => RR])org.parboiled2.Rule[j.
14 In,j.Out] <and>
```

Listing 1.5: Niewielki fragment (14 z 133 linii) błędu wygenerowanego przez bibliotekę *parboiled2*, który pochodzi z prezentacji Li Haoyi na temat *FastParse*[11].

1.3.7. FastParse

FastParse*FastParse*[12] to opracowana przez Li Haoyi, wysokowydajna biblioteka kombinatorów parserów dla Scali, zaprojektowana w celu uproszczenia tworzenia parserów tekstu strukturalnego. Umożliwia ona programistom definiowanie parserów rekurencyjnych, dzięki czemu nadaje się do parsowania języków programowania, formatów danych, takich jak JSON, czy DSL-i. Cechą charakterystyczną FastParse jest równowaga między użytecznością a wydajnością. Parsery są konstruowane poprzez łączenie mniejszych parserów za pomocą operatorów, takich jak `~` dla sekwencjonowania i `|` dla alternatyw, przy jednoczesnym zachowaniu czytelności zbliżonej do formalnych definicji gramatyki. Według dokumentacji[12], parsery *Fastparse* zajmują 1/10 kodu w porównaniu do ręcznie napisanego parsera rekurencyjnego. W porównaniu do narzędzi generujących parsery, takich jak *ANTLR* lub *Lex* i *Yacc*, implementacja nie wymaga żadnego specjalnego kroku kompilacji lub generowania kodu. To sprawia, że rozpoczęcie pracy z *Fastparse* jest znacznie łatwiejsze niż w przypadku bardziej tradycyjnych narzędzi do generowania parserów. Przykładowo, parser wyrażeń arytmetycznych może być zwięźle napisany, aby obsługiwać zagnieżdżone nawiasy, pierwszeństwo operatorów i raportowanie błędów w mniej niż 20 liniach kodu[13]. Biblioteka kładzie również nacisk na debugowanie, generując szczegółowe komunikaty o błędach, które wskazują dokładną lokalizację i przyczynę niepowodzeń parsowania, takich jak niedopasowane nawiasy lub nieprawidłowe tokeny.

1.3.8. Podsumowanie

1.3.9. Analiza ryzyka

Realizacja projektu wiąże się z szeregiem potencjalnych zagrożeń, które mogą wpłynąć na jego przebieg oraz ostateczny rezultat. Poniżej przedstawiono najważniejsze ryzyka

zidentyfikowane na etapie planowania, wraz z proponowanymi działaniami minimalizującymi ich wpływ:

- **Złożoność algorytmów parsingu:** Implementacja narzędzi takich jak lekser i parser od podstaw stanowi złożone wyzwanie wymagające zaawansowanej wiedzy z zakresu teorii kompilacji oraz algorytmiki. Może to prowadzić do opóźnień lub nawet uniemożliwić realizację projektu. W celu ograniczenia tego ryzyka przeprowadzono pogłębioną analizę zagadnień związanych z gramatykami formalnymi, automatami skończonymi oraz wyrażeniami regularnymi. Wykorzystano również istniejące rozwiązania jako punkt odniesienia w zakresie projektowania architektury systemu.

- **Wydajność rozwiązania:** Jednym z głównych założeń projektu jest stworzenie narzędzia o wyższej wydajności niż analogiczne rozwiązania dostępne w języku Python. Niespełnienie tego założenia mogłoby znacząco ograniczyć użyteczność biblioteki. W celu zapewnienia wysokiej efektywności obliczeniowej wybrano język Scala, a kluczowe elementy analizatora syntaktycznego realizowane są z wykorzystaniem makr kompilacyjnych, co pozwala na przeniesienie części obliczeń na etap kompilacji i znaczące przyspieszenie działania programu w czasie wykonywania.

- **Brak kompatybilności z popularnymi zintegrowanymi środowiskami programistycznymi (IDE):** Istniejące narzędzia do generowania analizatorów często nie oferują wsparcia dla współczesnych środowisk IDE, co utrudnia ich użycie i debugowanie. Projekt zakłada implementację biblioteki w czystym języku Scala, bez wykorzystania dedykowanego języka dziedzinowego (DSL), co umożliwia jej integrację z popularnymi narzędziami deweloperskimi wspierającymi Scalę, takimi jak IntelliJ IDEA czy Metals.

- **Wysoki próg wejścia:** Złożoność i różnorodność istniejących narzędzi do tworzenia parserów skutkuje trudnościami w ich opanowaniu. Projekt przewiduje zaprojektowanie przejrzystego i intuicyjnego interfejsu programistycznego (API) oraz przygotowanie kompleksowej dokumentacji technicznej. Działania te mają na celu obniżenie bariery wejścia i ułatwienie użytkownikom rozpoczęcia pracy z biblioteką.

- **Złożoność mechanizmów metaprogramowania w języku Scala:** W celu osiągnięcia zakładanej wydajności oraz elastyczności projekt wykorzystuje zaawansowane techniki metaprogramowania dostępne w języku Scala. Ich zastosowanie wymaga pogłębionej znajomości języka oraz jego systemu makr. W celu ograniczenia ryzyka wynikającego z potencjalnych trudności technicznych, przeprowadzono szczegółową analizę dokumentacji języka oraz ukończono specjalistyczny kurs z zakresu metaprogramowania w Scali[14].

- **Zbyt szeroki zakres funkcjonalny projektu:** Zakres planowanych funkcjonalności może przekraczać możliwości realizacyjne w ramach dostępnego czasu. Aby ograniczyć to ryzyko, projekt został podzielony na etapy według priorytetu wdrażania funkcji. W przypadku napotkania trudności możliwe będzie zakończenie projektu z działającym minimalnym zakresem funkcjonalnym (MVP), spełniającym podstawowe założenia.

Narzędzie	**Lex&Yacc**	**PLY/SLY**	**ANTLR**	**scala-bison**
Język implementacji	C	Python	Java	Scala (nad Bisonem)
Język użycia	regex, BNF, akcje w C	DSL	DSL oparty na EBNF	BNF, akcje w Scali
Wydajność	wysoka	niska	umiarkowana	wysoka
Łatwość użycia	średnia	umiarkowana	wysoka	średnia
Aktywne wsparcie	brak	nie	tak	nie
Diagnostyka błędów	słaba	średnia	dobra	słaba
Dokumentacja	dobra	średnia, nieaktualna	dobra	słaba
Popularność	wysoka	średnia	wysoka	niska
Integracja IDE	nieoficjalny plugin	ograniczona	oficjalny plugin	brak
Wsparcie do debugowania	brak	dobre	częściowe	dobre
Generowania kodu	nie	nie	tak	nie

Narzędzie	**Scala parser combinators**	**parboiled2**	**FastParse**	**ALPACA**
Język implementacji	Scala	Scala	Scala	Scala
Język użycia	DSL w Scali	DSL w Scali	DSL w Scali	Scala
Wydajność	wysoka	umiarkowana	wysoka	TODO
Łatwość użycia	niska	średnia	średnia	TODO
Aktywne wsparcie	nie	nie	tak	TODO
Diagnostyka błędów	dobra	niska	dobra	TODO
Dokumentacja	słaba	bardzo dobra	bardzo dobra	TODO
Popularność	średnia	niska	rosnąca	TODO
Integracja IDE	wsparcie dla Scali	wsparcie dla Scali	wsparcie dla Scali	TODO
Wsparcie do debugowania	dobre	dobre	dobre	TODO
Generowania kodu	nie	nie	nie	TODO

Tabela 1.1: Porównanie wybranych narzędzi do generowania lekserów i parserów

Rozdział 2

Zakres funkcjonalności

2.1. Wymagania funkcjonalne

- 1. Definicja parsera i gramatyki
  - Możliwość definiowania reguł gramatycznych za pomocą czystego języka Scala z obsługą wyrażeń regularnych i hierarchii reguł (bez DSL).
  - Generowanie drzewa składniowego: Automatyczne tworzenie struktury AST (Abstract Syntax Tree) z formatu EBNF.
  - Możliwość definiowania reguł rekurencyjnych.
  - Precyzyjne raportowanie błędów z lokalizacją w źródle, sugestiami napraw i trybem odzyskiwania (np. pomijanie błędnych tokenów).
  - Obsługa precedencji i asocjacji operatorów.
-
- 2. Integracja z narzędziami developerskimi
  - Obsługa podświetlania składni, autouzupełniania i debugowania w środowiskach takich jak IntelliJ lub VS Code.
- 3. Optymalizacje wydajnościowe
  - Obsługa dużych plików poprzez przetwarzanie danych strumieniowo.
  - Przechowywanie skompilowanych gramatyk w pamięci w celu przyspieszenia powtarzalnych operacji.
- 4. Rozszerzalność
  - Możliwość dołączania funkcji wykonywanych podczas parsowania (np. walidacja kontekstowa).
  - Podział implementacji na niezależne komponenty (np. lexer, parser).
  - Mechanizmy transformacji AST: struktury TreeMap oraz TreeTraverser.
-

2.2. Wymagania niefunkcjonalne

-
- 1. Wydajność
  - Czas parsowania zbliżony lub lepszy od czasów narzędzi *FastParse* i *Sly*.
  - Zużycie pamięci stałe, niezależne od rozmiaru parsowanego pliku.
- 2. Skalowalność
  - Obsługa dużych gramatyk.
  - Modularność.
- 3. Niezawodność
  - Testy regresyjne: Pokrycie kodu testami 90%, z automatycznym uruchamianiem po każdej zmianie.
- 4. Kompatybilność
  - Wsparcie dla IntelliJ i VS Code (Metals).
  - Systemy operacyjne: Działanie na Windows, Linux i macOS bez modyfikacji kodu.
- 5. Użyteczność
  - Obszerna dokumentacja z przykładami.
  - Narzędzia diagnostyczne, np. wizualizacja drzewa AST.
  - Weryfikacja poprawności gramatyk na poziomie typów.

2.3. Charakterystyka użytkowników

Alpaca jest skierowana do wszystkich zajmujących się aalizą składniową, lub gramatyczną, oraz tworzeniem języków formalnych, w szczególności języków programowania. W zależności od kontekstu można wyróżnić następujące grupy docelowe:

- **Programiści tworzący języki domenowe (DSL):** Specjaliści projektujący rozwiązania specyficzne dla danego problemu (np. konfiguratory[1], silniki reguł[2],), dla których kluczowa jest elastyczność definiowania składni. Dla tej grupy istotna jest także niezawodność oraz wydajność procesu parsowania.

[1]Konfigurator to specjalizowany język lub narzędzie umożliwiające deklaratywne definiowanie zachowania systemu lub aplikacji, najczęściej poprzez pliki konfiguracyjne. Przykłady to `docker-compose.yml` czy pliki konfiguracji serwisów CI/CD.

[2]Silnik reguł (ang. *rule engine*) to system przetwarzający zbiory warunków logicznych i wyzwalający odpowiednie akcje w oparciu o zdefiniowane reguły. Typowym zastosowaniem jest automatyzacja procesów biznesowych. Przykłady: Drools, CLIPS.

- **Studenci kierunków technicznych:** Osoby uczące się podstaw teorii kompilacji, które korzystają z narzędzi typu lekser/parser w celach dydaktycznych. Dla tej grupy kluczowe znaczenie ma intuicyjny interfejs programistyczny (API), rozbudowana dokumentacja oraz możliwość szybkiego uruchomienia przykładów bez konieczności ręcznej konfiguracji środowiska.

- **Nauczyciele akademiccy i prowadzący zajęcia laboratoryjne:** Osoby przygotowujące kursy dotyczące języków programowania, parserów, automatów i kompilatorów. *Alpaca* może służyć jako narzędzie wspierające zajęcia, umożliwiające praktyczną demonstrację działania gramatyk oraz parserów w sposób prostszy i szybszy niż za pomocą niskopoziomowych narzędzi, takich jak Lex i Yacc.

- **Entuzjaści języków programowania i narzędzi deweloperskich:** Osoby zainteresowane eksperymentowaniem z nowymi technologiami, tworzeniem i rozwojem własnych języków lub eksploracją działania analizatorów leksykalnych i składniowych. Ta grupa użytkowników ceni sobie możliwość rozszerzania i dostosowywania narzędzia do własnych potrzeb.

Uwzględnienie potrzeb i oczekiwań powyższych grup użytkowników stanowiło główny cel projektowania architektury systemu oraz definiowania jego funkcjonalności.

2.4. Scenariusze użytkowania i testowania

Poniższe scenariusze prezentują typowe przypadki użycia biblioteki *Alpaca*, wraz z towarzyszącymi im kryteriami poprawności oraz celami testowymi.

Scenariusz 1: Definicja parsera

Cel: Zdefiniowanie parsera dla prostego języka wyrażeń arytmetycznych.

Kroki:

1. Użytkownik definiuje leksemy (np. liczby, operatory arytmetyczne).

2. Tworzy nieterminale (np. `wyrazenie`, `iloczyn`) oraz odpowiadające im reguły gramatyczne (np. `potega = podstawa '**' wykladnik`).

3. Deklaruje funkcje ewaluacyjne dla odpowiednich produkcji (np. `pow(base, exponent)` dla potęgowania).

4. Uruchamia parser dla przykładowego ciągu wejściowego.

Oczekiwany rezultat: Parser generuje poprawne drzewo składniowe (AST) oraz zwraca wynik zgodny z semantyką definiowanej gramatyki.

Przykład testowy: `parse("2 + 2")` zwraca wartość `4` oraz drzewo składniowe reprezentujące operację dodawania z dwoma operandami liczbowymi.

Scenariusz 2: Obsługa błędów składniowych

Cel: Weryfikacja zachowania parsera w przypadku niepoprawnych danych wejściowych.

Kroki:

1. Użytkownik korzysta z przykładowej definicji parsera udostępnionej w dokumentacji lub przez prowadzącego.
2. Parser uruchamiany jest na niepoprawnym ciągu wejściowym (np. brakujący nawias zamykający).
3. System zwraca komunikat o błędzie z podaniem lokalizacji problemu oraz jego opisu.

Oczekiwany rezultat: System zgłasza czytelny komunikat błędu zawierający informacje o miejscu i charakterze błędu składniowego.

Scenariusz 3: Weryfikacja deterministyczności gramatyki

Cel: Ocena, czy zadana gramatyka jest deterministyczna i wolna od niejednoznaczności.

Kroki:

1. Użytkownik konstruuje złożoną gramatykę zawierającą alternatywne reguły.
2. Parser analizuje zachowanie dla potencjalnie niejednoznacznych ciągów wejściowych.

Oczekiwany rezultat: System wykrywa i zgłasza potencjalne konflikty składniowe, informując o możliwej niejednoznaczności gramatyki.

Scenariusz 4: Testowanie wydajności dla dużych wejść

Cel: Pomiar wydajności działania parsera w przypadku dużych danych wejściowych.

Kroki:

1. Parser uruchamiany jest na ciągu wejściowym o rozmiarze kilkunastu megabajtów.
2. Rejestrowany jest czas działania oraz poziom wykorzystania zasobów systemowych (pamięć operacyjna, CPU).

Oczekiwany rezultat: Parser przetwarza dane w akceptowalnym czasie, nie generując błędów oraz nie przekraczając założonego limitu zużycia pamięci.

Bibliografia

[1] M. E. Lesk i E. Schmidt. *Lex: A lexical analyzer generator*. T. 39. Bell Laboratories Murray Hill, NJ, 1975.

[2] S. C. Johnson i in. *Yacc: Yet another compiler-compiler*. T. 32. Bell Laboratories Murray Hill, NJ, 1975.

[3] D. Beazley. *PLY (Python Lex-Yacc)*. 2005. URL: https://www.dabeaz.com/ply/ply.html (term. wiz. 19. 03. 2025).

[4] D. Beazley. *SLY (Sly Lex-Yacc)*. 2016. URL: https://sly.readthedocs.io/en/latest/sly.html (term. wiz. 19. 03. 2025).

[5] D. Beazley. *SLY Github*. URL: https://github.com/dabeaz/sly (term. wiz. 19. 03. 2025).

[6] T. Parr, P. Wells, R. Klaren, L. Craymer, J. Coker, S. Stanchfield, J. Mitchell i C. Flack. *What's ANTLR*. 2004.

[7] A. Moors, F. Piessens i M. Odersky. „Parser combinators in Scala". W: *CW Reports* (2008).

[8] *scala-parser-combinators Getting Started*. URL: https://github.com/scala/scala-parser-combinators/blob/main/docs/Getting_Started.md (term. wiz. 04. 04. 2025).

[9] J. Boyland i D. Spiewak. „Tool paper: ScalaBison recursive ascent-descent parser generator". W: *Electronic Notes in Theoretical Computer Science* 253.7 (2010).

[10] A. A. Myltsev. „Parboiled2: A macro-based approach for effective generators of parsing expressions grammars in Scala". W: *arXiv preprint arXiv:1907.03436* (2019).

[11] L. Haoyi. *sfscala.org: Li Haoyi, FastParse: Fast, Programmable, Modern Parser-Combinators in Scala*. 2015.

[12] *FastParse Getting Started*. URL: https://com-lihaoyi.github.io/fastparse/#GettingStarted (term. wiz. 04. 04. 2025).

[13] L. Haoyi. *FastParse. Fast, Modern Parser Combinators*. URL: https://www.lihaoyi.com/post/slides/FastParse.pdf (term. wiz. 18. 04. 2025).

[14] D. Ciocîrlan. *Scala Macros and Metaprogramming*. URL: https://rockthejvm.com/courses/scala-macros-and-metaprogramming (term. wiz. 17. 05. 2025).

Spis rysunków

Spis tabel

Spis algorytmów

Spis listingów