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, 2026

I do see the beauty in the rules, the invisible code of chaos hiding behind the menacing face of order.

— Elliot Alderson, *Mr. Robot (episode eps2.0_unm4sk-pt1.tc)*

Abstrakt

Analizatory leksykalne i składniowe stanowią fundamentalne komponenty procesu kompilacji, realizując fazy transformacji ciągu znaków wejściowych na strumień tokenów oraz weryfikacji zgodności strukturalnej z gramatyką. Istniejące narzędzia do konstruowania tych analizatorów wykazują ograniczenia w kontekście nowoczesnych języków programowania oraz narzędzi deweloperskich. Popularne generatory kodu takie jak Lex/Yacc i ANTLR wymagają zewnętrznych etapów budowania i znajomości języków domenowych, podczas gdy biblioteki interpretowane jak PLY i SLY cechują się słabą wydajnością i ograniczonym bezpieczeństwem typów. Kombinatory parserów oferują elastyczność w opisie i modyfikacji składni języka, jednak często wiążą się z dodatkowym narzutem wykonania zmniejszającym wydajność w porównaniu do podejść generacyjnych. Praca przedstawia bibliotekę ALPACA (Another Lexer Parser And Compiler Alpaca) — narzędzie opracowane w Scali 3, które korzystając z możliwości metaprogramowania w czasie kompilacji, łączy wydajność generatorów kodu z użytecznością bibliotek. Teza badawcza postuluje, że wykorzystanie makr Scali 3, mechanizmu cytatów i wstawek oraz typów rafinowanych umożliwia konstrukcję lekserów i parserów o trzech właściwościach. Po pierwsze, zapewniają one wydajność zbliżoną do innych narzędzi. Po drugie, oferują interfejs programistyczny niezależny od zewnętrznych języków domenowych i w pełni zintegrowany z systemem typów Scali. Po trzecie, dostarczają diagnostykę błędów w czasie kompilacji. Metodologia wykorzystuje API refleksji TASTy do programatycznego generowania klas anonimowych implementujących logikę tokenizacji i parsowania w trakcie kompilacji. Specyfikacja analizy leksykalnej opiera się na wyrażeniach regularnych, ograniczając próg wejścia i wspierając praktyczną adopcję narzędzia. Analiza składniowa implementuje algorytmy konstrukcji parserów LR(1) poprzez wykorzystanie automatu ze stosem, którego struktura stanowa jest generowana z użyciem funkcji do wyznaczania zbiorów elementów LR(1) oraz przejść między stanami automatu. Implementacja rozwiązuje ograniczenia JVM poprzez techniki fragmentacji metod, umożliwiając kompilację złożonych gramatyk. Akcje semantyczne zachowują bezpieczeństwo typów poprzez transformacje AST i przepisywanie referencji między etapami kompilacji. Kluczowe wkłady techniczne obejmują deklaratywną specyfikację gramatyki zintegrowaną z dopasowaniem wzorców i systemem typów Scali, co stanowi implementację wewnętrznego języka domenowego opartego na składni języka. Typy rafinowane zapewniają statycznie weryfikowany dostęp do pól tokenów z pełnym wsparciem IDE, obejmującym autouzupełnianie, podpowiedzi typów i nawigację do definicji. System wspiera także rozwiązywanie konfliktów poprzez relacje precedencji oraz walidację gramatyki w czasie kompilacji. Ograniczenia precedencji modelowane są jako graf skierowany weryfikowany pod kątem acykliczności metodą DFS z kolorowaniem węzłów; wykryte cykle raportowane są jako niejednoznaczności specyfikacji. Reprezentacja grafowa umożliwia wnioskowanie o relacjach przechodnich, co pozwala na automatyczne rozwiązywanie konfliktów wynikających z pośrednich zależności precedencji. Walidacja empiryczna porównuje bibliotekę ALPACA z alternatywnymi narzędziami w testach wydajnościowych obejmujących parsowanie wyrażeń arytmetycznych i plików w formacie JSON. Wyniki wskazują na konkurencyjną wydajność dla struktur iteracyjnych, identyfikując wrażliwość na obciążenia zdominowane przez tokeny ignorowane. W obecnej implementacji białe znaki przetwarzane są analogicznie do tokenów właściwych, co prowadzi do degradacji wydajności w przypadkach patologicznych, takich jak głęboko wcięte struktury JSON z dużą liczbą spacji przy niewielkiej ilości treści semantycznej. Ograniczenie to wskazano jako kierunek dalszych optymalizacji. Wyniki wspierają tezę, że metaprogramowanie w czasie kompilacji może zapewniać wysoką wydajność przy jednoczesnym zwiększeniu użyteczności i integracji z IDE.

Słowa kluczowe: metaprogramowanie, Scala 3, makra, analizator leksykalny, parser składniowy, parser LR(1), quote-splice, typy rafinowane, generowanie kodu, refleksja TASTy, bezpieczeństwo typów, optymalizacja kompilacji, integracja IDE, gramatyki.

Abstract

Lexical and syntactic analyzers are fundamental components of the compilation process, transforming input character sequences into token streams and verifying structural conformance to a grammar. Existing tools for constructing these analyzers face significant limitations regarding modern programming languages and the features offered by Integrated Development Environments (IDEs). Popular code generators, such as Lex/Yacc and ANTLR, require external build steps and familiarity with external domain-specific languages (DSLs), while interpreted libraries like PLY and SLY are characterized by poor performance and limited type safety. Conversely, parser combinators offer flexibility in grammar specification but often introduce runtime overhead that degrades performance compared to generative approaches. This thesis introduces ALPACA (Another Lexer Parser And Compiler Alpaca), a library developed in Scala 3 that bridges the gap between the performance of code generators and the usability of libraries by leveraging compile-time metaprogramming. The research posits that utilizing Scala 3 macros, the quote-and-splice mechanism, and refined types enables the construction of lexers and parsers with three key properties. First, parsing performance comparable to specialized generative tools. Second, a programming interface decoupled from external DSLs and fully integrated with the Scala type system. Finally, comprehensive compile-time error diagnostics. The methodology employs the TASTy reflection API to programmatically generate anonymous classes implementing tokenization and parsing logic during compilation. Lexical specifications are expressed using regular expressions, leveraging a widely adopted mechanism to reduce the learning burden and improve practical adoption. Syntactic analysis implements LR(1) parser construction algorithms by employing a pushdown automaton whose state structure is generated using the closure and goto functions to compute LR(1) item sets and transitions between automaton states. The implementation addresses critical JVM constraints through method fragmentation techniques, enabling the compilation of complex grammars. Semantic actions maintain type safety through AST transformations and cross-stage reference rewriting. Key technical contributions include a declarative grammar specification integrated with Scala's pattern matching and type system, effectively implementing an internal DSL. Refined types provide statically verified access to token fields with full IDE support, including autocomplete, type hints, and go-to-definition navigation. The system additionally supports precedence-based conflict resolution and compile-time grammar validation. Precedence constraints are modeled as a directed graph validated for acyclicity using depth-first search with node coloring; cycles are reported as specification ambiguities. The graph representation further enables transitive inference, allowing automatic resolution of conflicts implied by indirect precedence paths. Evaluation compares ALPACA with alternative tools using benchmarks for arithmetic expression parsing and JSON parsing. Results indicate competitive performance on iterative inputs, while identifying a performance sensitivity in workloads dominated by ignored tokens. In the current implementation, whitespace is processed equivalently to regular tokens, leading to degradation on pathological cases such as deeply indented JSON with large volumes of whitespace and limited semantic content. This limitation is identified as a primary target for future optimization. Overall, the results support the thesis claim that compile-time metaprogramming can deliver high performance while improving usability and IDE integration.

Keywords: metaprogramming, Scala 3, macros, lexical analyzer, LR(1) parser, quote-splice mechanisms, refined types, code generation, TASTy reflection, type safety, compilation optimization, IDE integration, grammars.

Spis treści

Rozdział 1

Cel pracy i wizja projektu

1.1. Charakterystyka problemu

Analizatory leksykalne (ang. *lexers*) i składniowe (ang. *parsers*) stanowią fundamentalne komponenty procesu kompilacji, realizując odpowiednio fazę analizy leksykalnej i syntaktycznej [1]. Analizator leksykalny segmentuje ciąg znaków wejściowych na strumień tokenów (leksemów) zgodnie z regułami języka regularnego [2], podczas gdy analizator składniowy weryfikuje zgodność strumienia tokenów z gramatyką bezkontekstową języka, konstruując drzewo składni abstrakcyjnej (AST, ang. *Abstract Syntax Tree*) [1].

1.1.1. Podstawy teoretyczne

Analiza leksykalna i składniowa opiera się na teorii języków formalnych, zapoczątkowanej przez prace Noama Chomsky'ego [3]. W hierarchii Chomsky'ego języki dzieli się na cztery klasy według mocy wyrazu gramatyk je generujących. Analizatory leksykalne operują na językach regularnych (typ 3), które są rozpoznawane przez automaty skończone [2], podczas gdy parsery składniowe obsługują języki bezkontekstowe (typ 2), rozpoznawane przez automaty ze stosem [1].

Wyrażenia regularne są notacją deklaratywną dla języków regularnych i można je mechanicznie przekształcić w automaty skończone za pomocą konstrukcji Thompsona [4]. Automaty deterministyczne (DFA) gwarantują liniową złożoność czasową rozpoznawania $O(n)$, podczas gdy niedeterministyczne (NFA) mogą wymagać przeszukiwania z nawrotami (ang. *backtracking*).

Gramatyki bezkontekstowe (CFG) definiują strukturę syntaktyczną języków programowania. Parsery dla CFG dzielą się na dwie główne kategorie: parsery zstępujące rekurencyjnie (ang. *top-down*), takie jak LL(k) [5], oraz parsery wstępujące rekurencyjnie (ang. *bottom-up*), takie jak LR(k) [6]. Wybór klasy parsera determinuje kompromisy między mocą wyrazu gramatyki, złożonością implementacji oraz jakością komunikatów błędów.

1.2. Teza i pytania badawcze

W niniejszej pracy przyjęto tezę, zgodnie z którą wykorzystanie metaprogramowania w języku Scala 3 (makra kompilacyjne, typy rafinowane) umożliwia konstrukcję systemu generującego analizatory leksykalne i składniowe charakteryzujących się następującymi właściwościami:

- wydajność — czas parsowania porównywalny z narzędziami opartymi na generacji kodu (ANTLR, Yacc), przewyższający biblioteki interpretowane (PLY, SLY),
- użyteczność — interfejs programistyczny (API) niezależny od dedykowanego DSL, zintegrowany z systemem typów Scali i wspierany przez standardowe narzędzia IDE,
- diagnostyka błędów — komunikaty błędów generowane w czasie kompilacji (dla błędów gramatyki) oraz w czasie parsowania (dla błędów składniowych), zawierające kontekst syntaktyczny.

W ramach weryfikacji tezy sformułowano następujące pytania badawcze:

- Czy możliwe jest osiągnięcie wydajności zbliżonej do generatorów kodu przy zachowaniu elastyczności bibliotek kombinatorów poprzez zastosowanie metaprogramowania?
- W jakim stopniu wykorzystanie typów rafinowanych w Scali 3 wpływa na bezpieczeństwo typów i komfort pracy z wygenerowanym parserem?
- Jakie ograniczenia maszyny wirtualnej Java (JVM) wpływają na proces generacji kodu w czasie kompilacji i jak można je efektywnie niwelować?

Celem pracy jest zaprojektowanie i zaimplementowanie narzędzia *ALPACA* (*Another Lexer Parser And Compiler Alpaca*) w języku Scala, które implementuje funkcjonalności powszechnie stosowane w budowie analizatorów leksykalnych i składniowych, weryfikując postawioną tezę.

1.3. Motywacja projektu

Istniejące narzędzia do konstrukcji analizatorów leksykalnych i składniowych wykazują szereg ograniczeń utrudniających ich zastosowanie w kontekście nowoczesnych języków programowania oraz środowisk deweloperskich. Identyfikacja tych ograniczeń stanowiła punkt wyjścia dla projektu *ALPACA*.

Projekt *ALPACA* stanowi narzędzie do generowania lekserów i parserów w języku Scala, łączące zalety istniejących rozwiązań poprzez:

- Połączenie wydajności generatorów kodu z użytecznością bibliotek, czyli wykorzystanie makr kompilacyjnych Scali 3, co pozwala przenieść część obliczeń na etap kompilacji, zachowując interfejs programistyczny zintegrowany z systemem typów języka.
- Generowanie komunikatów błędów w oparciu o kontekst parsera LR(1).
- Natywną integrację ze środowiskami IDE, gdyż implementacja w czystym języku Scala eliminuje konieczność stosowania dedykowanych pluginów, wykorzystując istniejące wsparcie dla języka (IntelliJ IDEA, Metals).

Proponowane rozwiązanie łączy nowoczesne podejście technologiczne z praktycznym zastosowaniem w edukacji i programowaniu. Może ono służyć jako narzędzie dydaktyczne,

ułatwiając naukę teorii kompilacji, w pracach badawczych, a także jako kompleksowe narzędzie do tworzenia praktycznych rozwiązań.

1.4. Przegląd istniejących rozwiązań

Narzędzia do konstrukcji analizatorów leksykalnych i składniowych można sklasyfikować według strategii implementacyjnej na trzy główne kategorie: generatory kodu, biblioteki interpretowane oraz kombinatory parserów.

1.4.1. Generatory kodu

Generatory kodu transformują deklaratywne specyfikacje gramatyk w kod źródłowy parsera w języku docelowym. Proces ten odbywa się przed kompilacją programu głównego i wymaga dodatkowego narzędzia w procesie budowania (ang. *build chain*).

Lex i Yacc

Lex [7] i *Yacc* [8] 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. Narzędzia te wymuszają znajomość dedykowanej składni specyfikacji gramatyk, co utrudnia rozpoczęcie pracy dla początkujących użytkowników.

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;
```

```
25 }
26 else if ($5 == tANDOP) {
27     $5 = 1;
28 }
29 $$ = NEW_OP_ASGN1($1, $5, args);
30 fixpos($$, $1);
31 /*%
32 $$ = dispatch2(aref_field, $1, escape_Qundef($3));
33 $$ = dispatch3(opassign, $$, $5, $6);
34 %*/
35 }
```

Listing 1.1. Fragment definicji parsera Ruby z wykorzystaniem technologii Yacc.

ANTLR

ANTLR [9] to 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.

1.4.2. Biblioteki interpretowane

Biblioteki interpretowane definiują gramatyki jako struktury danych w języku bazowym. Parser jest wykonywany w czasie działania programu, co eliminuje krok generacji kodu, ale wprowadza narzut wydajnościowy.

PLY i SLY

PLY [10] i jego nowszy odpowiednik *SLY* [11] 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 tworzenia analizatora leksykalnego lub składniowego. Mechanizm refleksji wykorzystywany przez bibliotekę *SLY* (inspekcja nazw metod i typów) powoduje generowanie ostrzeżeń przez analizatory statyczne środowiska PyCharm. Ponadto należy zaznaczyć, iż autor projektu informuje o braku dalszego rozwoju tych narzędzi [12].

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

Dekorator `@_()` definiuje wzorzec dopasowania dla produkcji. Argumenty w cudzysłowie są traktowane jako literały, podczas gdy identyfikatory bez cudzysłowu odnoszą się do innych nieterminali.

Konwencja nazewnictwa metod określa typ zwracany przez produkcję. Parser automatycznie identyfikuje wszystkie metody o danej nazwie jako alternatywne produkcje dla tego nieterminala. Mechanizm ten eliminuje potrzebę jawnej deklaracji reguł, ale utrudnia śledzenie struktury gramatyki.

W krotce `precedence` definiowane jest 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 we wzorcu dopasowania w adnotacji. Jeśli zdefiniowany jest więcej niż jeden element, dodawany jest numer do akcesora, np. `expr1` jest odwołaniem do drugiego wyrażenia `expr`.

```
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):
    raise NotImplementedError

@_('IF "(" condition ")" block ELSE block')
def instruction(self, p: YaccProduction):
    raise NotImplementedError

@_('expr EQUAL expr')
def condition(self, p: YaccProduction):
    args = [p.expr0, p.expr1]
    raise NotImplementedError
```

Listing 1.2. Fragment definicji parsera w Pythonie, wykorzystujący bibliotekę SLY.

Komunikaty błędów w bibliotece *SLY* nie zawierają informacji o kontekście syntaktycznym ani sugestii poprawek, co obrazuje przykład 1.3, który po uruchomieniu informuje

użytkownika błędem z fragmentu kodu 1.4. Okazuje się, że problemem był brak atrybutu `ignore\_comment` w definicji `Lexer`.

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

Listing 1.3. Fragment niedziałającego kodu w Pythonie, wykorzystujący bibliotekę SLY.

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

Listing 1.4. Przykładowy komunikat błędu w bibliotece *SLY*.

1.4.3. Kombinatory parserów

Kombinatory parserów to funkcje wyższego rzędu konstruujące złożone parsery z prostszych komponentów. Podejście to łączy elastyczność bibliotek z czytelną składnią zbliżoną do notacji BNF.

Scala parser combinators

Biblioteka *Scala parser combinators* [13] była popularnym sposobem na tworzenie parserów, lecz jak stwierdzono w samej 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 stanowić istotną przeszkodę, a standardowa dokumentacja nie jest zbyt pomocna” [14].

ScalaBison

Z podsumowania artykułu na temat *ScalaBison* [15] 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*.

parboiled2

parboiled2 [16] 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. Ze względu na skomplikowany i nieintuicyjny DSL, bariera wejścia dla nowych użytkowników jest wysoka. Zgodnie z przykładem 1.5, komunikaty błędów nie zawierają informacji o kontekście syntaktycznym oraz nie sugerują możliwych poprawek (problem z implementacją wynika jedynie z różnic w liczbie parametrów funkcji).

```
[error] /Users/haoyi/Dropbox (Personal)/Workspace/scala-js-book/scalatexApi/src/main/scala/scalatex/stages/Parser.scala:60: overloaded
method value apply with alternatives:
[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.
Text, scalatex.stages.Ast.Chain, Int, scalatex.stages.Ast.Block) => RR)(implicit j: org.parboiled2.support.ActionOps.SJoin[shapeless.::[I,
shapeless.::[J,shapeless.::[K,shapeless.::[L,shapeless.::[M,shapeless.::[N,shapeless.::[O,shapeless.::[P,shapeless.::[Q,shapeless.::[R,
shapeless.::[S,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[(I, J, K, L, M, N, O, P, Q, R, S, 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.In,j.Out] <and>
[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,
scalatex.stages.Ast.Chain, Int, scalatex.stages.Ast.Block) => RR)(implicit j: org.parboiled2.support.ActionOps.SJoin[shapeless.::[J,
shapeless.::[K,shapeless.::[L,shapeless.::[M,shapeless.::[N,shapeless.::[O,shapeless.::[P,shapeless.::[Q,shapeless.::[R,shapeless.::[S,
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,
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.
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* [17].

FastParse

FastParse [18] 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 [18], 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 [19].

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.

1.4.4. Analiza porównawcza

Tabela 1.1 zestawia główne cechy analizowanych narzędzi. Widoczny jest kompromis między wydajnością a użytecznością: generatory kodu (Lex/Yacc, ANTLR) osiągają wysoką wydajność, ale wymagają dodatkowego kroku kompilacji i nauki DSL. Biblioteki kombinatorów (FastParse, parboiled2) oferują interfejs zintegrowany z językiem bazowym, ale kosztem spadku wydajności związanej z interpretacją reguł w czasie wykonania.
Żadne z analizowanych rozwiązań nie łączy jednocześnie:

- wysokiej wydajności (generacja kodu w czasie kompilacji),
- interfejsu API zintegrowanego z systemem typów języka,
- komunikatów błędów zawierających kontekst syntaktyczny,
- natywnej integracji ze środowiskami IDE bez dedykowanych pluginów.

Luka ta stanowi motywację dla projektu *ALPACA*, który wykorzystuje makra kompilacyjne Scali 3 do osiągnięcia tych celów jednocześnie.

1.5. Ograniczenia i zakres pracy

Niniejsza praca koncentruje się na implementacji parsera LR(1) oraz analizatora leksykalnego wykorzystującego wyrażenia regularne. Następujące aspekty wykraczają poza zakres pracy:

- System generuje kanoniczne stany LR(1) bez minimalizacji do LALR(1), co może prowadzić do większych tablic akcji. Implementacja minimalizacji stanowi potencjalny kierunek przyszłych badań.
- Ewaluacja empiryczna w kontekście dydaktycznym, czyli weryfikacja użyteczności systemu w środowisku akademickim (badanie z udziałem studentów) wykracza poza zakres pracy i stanowi kierunek przyszłych badań.

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
Generowanie 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	wysoka
Łatwość użycia	niska	średnia	średnia	wysoka
Aktywne wsparcie	nie	nie	tak	tak
Diagnostyka błędów	dobra	niska	dobra	dobra
Dokumentacja	słaba	bardzo dobra	bardzo dobra	dobra
Popularność	średnia	niska	rosnąca	niska
Integracja IDE	wsparcie dla Scali	wsparcie dla Scali	wsparcie dla Scali	wsparcie dla Scali
Wsparcie do debugowania	dobre	dobre	dobre	dobre
Generowanie kodu	nie	nie	nie	tak

Tabela 1.1. Porównanie wybranych narzędzi do generowania analizatorów leksykalnych i składniowych.

Rozdział 2

Metaprogramowanie w Scali 3

Niniejszy rozdział stanowi wprowadzenie do systemów metaprogramowania ze szczególnym uwzględnieniem mechanizmów dostępnych w języku Scala 3. Zrozumienie tych koncepcji jest kluczowe dla dalszej części pracy, ponieważ metaprogramowanie stanowi fundament technologiczny projektu *ALPACA*. To właśnie dzięki tym technikom możliwe było zrealizowanie głównego celu projektu — generowania wydajnych lekserów i parserów w czasie kompilacji, przy zachowaniu pełnego bezpieczeństwa typów i bez konieczności stosowania zewnętrznych narzędzi generujących kod źródłowy.

2.1. Wprowadzenie

Scala 3, znana również jako Dotty, wprowadza całkowicie przeprojektowany system metaprogramowania, stanowiący fundamentalną zmianę w stosunku do eksperymentalnych makr dostępnych w Scali 2 [20, 21]. Metaprogramowanie w Scali 3 zostało zaprojektowane z naciskiem na bezpieczeństwo typów, przenośność oraz skalowalność, umożliwiając twórcom oprogramowania generowanie i analizowanie kodu w czasie kompilacji przy zachowaniu pełnej ekspresywności języka [22, 23]. W przeciwieństwie do poprzedniego systemu, który eksponował wewnętrzne mechanizmy kompilatora i był źródłem problemów z kompatybilnością między wersjami [24], nowy system metaprogramowania jest zaprojektowany jako stabilny i przenośny interfejs programistyczny.

Podstawą teoretyczną systemu metaprogramowania w Scali 3 jest programowanie wieloetapowe (ang. *multi-stage programming*), paradygmat pozwalający na odróżnienie różnych etapów wykonania programu [25, 24]. W tym modelu kod może być wykonywany w różnych fazach: w czasie kompilacji (ang. *compile-time*) lub w czasie wykonania (ang. *runtime*) [25]. Rozdzielenie tych faz pozwala na przeniesienie obliczeń z czasu wykonania do czasu kompilacji, co potencjalnie eliminuje narzut wykonania i umożliwia wcześniejszą detekcję błędów.

2.1.1. Cytaty i wstawki

Kluczowymi koncepcjami w systemie metaprogramowania Scali 3 są cytaty (ang. *quotes*) i wstawki (ang. *splices*) [26, 27]. Cytaty, oznaczane jako `'{...}`, służą do opóźnienia wykonania kodu i traktowania go jako danych [28]. Wstawki, oznaczane jako `${...}`, pozwalają na ocenę wyrażenia generującego kod i wstawienie wyniku do otaczającego kontekstu [28, 29].

```scala
inline def square(x: Int): Int = ${ squareImpl('x) }

def squareImpl(x: Expr[Int])(using Quotes): Expr[Int] = '{
  val squared = $x * $x
  squared
}

// usage: square(3) → will be expanded to: val squared = 3 * 3; squared
```

Listing 2.1. Prosty przykład ilustrujący podstawowe wykorzystanie cytatów i wstawek w makrach.

W przykładzie 2.1 zaobserwować można następujące zależności.

- `'x` tworzy cytat (ang. *quote*) z wyrażenia `x`, opóźniając jego wykonanie,
- `'{...}` tworzy blok kodu jako dane, które będzie wstawione w miejscu wywołania makra,
- `$x` wstawia (ang. *splice*) wartość cytatu do nowego kontekstu.

Formalna semantyka tych konstrukcji została przedstawiona w pracy Stuckiego, Brachthäusera i Odersky'ego [27], gdzie cytaty i wstawki są traktowane jako prymitywne formy w typowanych drzewach składniowych (ang. *typed abstract syntax trees*). Autorzy dowodzą, że system zachowuje bezpieczeństwo typów oraz higieniczność, zapewniając, że wygenerowany kod nie może przypadkowo powiązać identyfikatorów z niewłaściwymi zmiennymi [27].

2.1.2. Bezpieczeństwo międzyetapowe

Scala 3 gwarantuje bezpieczeństwo międzyetapowe (ang. *cross-stage safety*) poprzez sprawdzanie poziomów etapowania w czasie kompilacji [24, 27]. Zmienne lokalne mogą być używane tylko na tym samym poziomie etapowania, na którym zostały zdefiniowane, co zapobiega dostępowi do zmiennych, które jeszcze nie istnieją lub już nie są dostępne [24].

```scala
def unsafeQuote(using Quotes): Expr[Int] =
  val localVar = 42
  '{ localVar }  // error: access to value localVar from wrong staging
    level
```

Listing 2.2. Przykład naruszenia bezpieczeństwa międzyetapowego (kod nie kompiluje się).

W przykładzie 2.2 kompilator wykryje ten błąd i zgłosi komunikat: `error: access to value localVar from wrong staging level`. Aby poprawnie odnieść się do wartości z otaczającego kontekstu, należy użyć mechanizmu `Expr.apply`.

```scala
def safeQuote(using Quotes): Expr[Int] =
  val localVar = 42
  Expr(localVar)  // OK: local variable is quoted
```

Listing 2.3. Poprawne przeniesienie wartości między etapami.

System również zapewnia, że typy generyczne używane w wyższym poziomie etapowania niż ich definicja wymagają instancji klasy typu `Type[T]`, która niesie reprezentację typu niepoddaną wymazywaniu (ang. *type erasure*) [24]. To podejście rozwiązuje problem

wymazywania typów generycznych w JVM, zachowując informację o typach potrzebną w kolejnych etapach kompilacji.

2.2. Mechanizmy metaprogramowania w Scali 3

Przedstawione powyżej podstawy teoretyczne znajdują bezpośrednie zastosowanie w praktycznych mechanizmach metaprogramowania oferowanych przez język Scala 3, które zostaną omówione w niniejszej sekcji.

2.2.1. Definicje inline

Najprostszym narzędziem metaprogramowania jest modyfikator `inline` [30]. Gwarantuje on, że wywołanie oznaczonej nim metody lub wartości zostanie w całości wstawione w miejscu wywołania (ang. *inlining*) podczas kompilacji. Jest to instrukcja dla kompilatora, a nie tylko sugestia, jak w niektórych innych językach [31].

```
inline def max(x: Int, y: Int): Int = inline if x > y then x else y

// usage: max(3, 5)
// will be expanded to: if 3 > 5 then 3 else 5
// after constant folding: 5
```

Listing 2.4. Użycie modyfikatora inline dla optymalizacji.

Modyfikator `inline` różni się od zwykłych funkcji tym, że **gwarantuje** wstawienie kodu, podczas gdy standardowe funkcje mogą być zinlinowane przez kompilator jako optymalizacja, ale nie muszą.

2.2.2. Makra oparte na wyrażeniach

Makra w Scali 3 są zdefiniowane jako metody `inline` zawierające wstawkę najwyższego poziomu (ang. *top-level splice*) [32, 33], czyli taki, który nie jest zagnieżdżony w żadnym cytacie (ang. *quote*) i jest wykonywany w czasie kompilacji [25, 32].

Typ `Expr[T]` reprezentuje wyrażenie Scali o typie `T` jako typowane drzewo składniowe [28, 33]. Makra manipulują wartościami typu `Expr[T]`, transformując je lub generując nowe wyrażenia [33]. Ta reprezentacja gwarantuje bezpieczeństwo typów na poziomie języka metaprogramowania [28].

```
inline def showType[T](x: T): String = ${ showTypeImpl('x) }

def showTypeImpl[T: Type](x: Expr[T])(using Quotes): Expr[String] =
  import quotes.reflect.*
  val tpe = TypeRepr.of[T]
  Expr(tpe.show)

// usage: showType(42)          // "scala.Int"
// usage: showType("hello")     // "java.lang.String"
```

Listing 2.5. Makro generujące kod inspekcji typu.

W przykładzie 2.5 makro `showType` wykorzystuje refleksję TASTy (sekcja 2.2.4) do uzyskania reprezentacji typu w czasie kompilacji i wygenerowania kodu zwracającego jego nazwę.

2.2.3. Dopasowanie wzorców w cytatach kodu

Scala 3 wspiera analizę kodu poprzez dopasowanie wzorców w cytatach kodu (ang. *quote pattern matching*) [24, 27]. Mechanizm ten pozwala na dekonstrukcję kawałków kodu i ekstrakcję podwyrażeń [27].

Stucki, Brachthäuser i Odersky [27] wprowadzają wzorce wiążące (ang. *bind patterns*) postaci `$x` oraz wzorce HOAS (ang. *Higher-Order Abstract Syntax*) postaci `$f(y)`, które pozwalają na ekstrakcję podwyrażeń potencjalnie zawierających zmienne z zewnętrznego kontekstu. System gwarantuje, że ekstrahowane wyrażenia są zamknięte względem definicji wewnątrz wzorca, zapobiegając wyciekom zakresu.

```
inline def optimize(x: Int): Int = ${ optimizeImpl('x) }

def optimizeImpl(x: Expr[Int])(using Quotes): Expr[Int] = x match
  case '{ 0 + $y }       => y  // 0 + y → y
  case '{ $y + 0 }       => y  // y + 0 → y
  case '{ 1 * $y }       => y  // 1 * y → y
  case '{ $y * 1 }       => y  // y * 1 → y
  case '{ 0 * $y }       => '{ 0 }  // 0 * y → 0
  case '{ $x + ($y + $z) } => '{ $x + $y + $z }  // reassociation
  case _ => x  // no optimization

// usage:
// optimize(0 + 5)      // 5
// optimize(3 * 1)      // 3
// optimize(0 * 100)    // 0
```

Listing 2.6. Przykład dopasowania wzorców kodu: optymalizacja wyrażeń algebraicznych poprzez dopasowanie wzorców.

Makro `optimize` rozpoznaje wzorce wyrażeń arytmetycznych i zastępuje je uproszczonymi wersjami w czasie kompilacji, eliminując zbędne operacje.

2.2.4. Refleksja TASTy

Dla przypadków wymagających głębszej analizy kodu, Scala 3 oferuje API refleksji TASTy [28, 34]. TASTy (ang. *Typed Abstract Syntax Trees*) jest binarnym formatem serializacji typowanych drzew składniowych używanym przez kompilator Scali 3 [24].

API refleksji dostarcza szczegółowy widok na strukturę kodu, włączając typy, symbole oraz pozycje w kodzie źródłowym. Jest dostępne poprzez obiekt `reflect` zdefiniowany w typie `Quotes`, który jest przekazywany kontekstualnie do makr [28, 34]. System refleksji TASTy definiuje następującą hierarchię typów:

- **Tree** — podstawowy typ reprezentujący węzeł drzewa składni
- **Term** — wyrażenia (np. wywołania funkcji, literały)
- **TypeTree** — reprezentacje typów w drzewie składni
- **Symbol** — symbole (definicje klas, metod, zmiennych)
- **TypeRepr** — reprezentacje typów (niezależne od drzewa)

Makro `inspectFields` wykorzystuje refleksję TASTy do ekstrakcji nazw pól klasy przypadku w czasie kompilacji, co pozwala na generowanie kodu specyficznego dla struktury typu bez ręcznej specyfikacji.

```
inline def inspectFields[T]: List[String] = ${ inspectFieldsImpl[T] }

def inspectFieldsImpl[T: Type](using Quotes): Expr[List[String]] =
  import quotes.reflect.*

  val tpe = TypeRepr.of[T]
  val fields = tpe.typeSymbol.declaredFields.map(_.name)

  Expr(fields)

// usage: case class Person(name: String, age: Int, city: String)
//        inspectFields[Person]  // List("name", "age", "city")
```

Listing 2.7. Przykład użycia refleksji TASTy: inspekcja struktury klasy przypadku za pomocą refleksji TASTy.

2.3. Porównanie z innymi systemami metaprogramowania

System metaprogramowania Scali 3 czerpie inspiracje z innych języków, ale wprowadza własne innowacje w zakresie bezpieczeństwa typów i ergonomii.

2.3.1. Makra w Lisp i Scheme

Język Lisp [35] był pionierem w dziedzinie metaprogramowania, wprowadzając koncepcję makr jako transformacji list reprezentujących kod. Kluczową różnicą między makrami Lisp a Scali 3 jest sposób reprezentacji kodu. Makra w Lisp operują na nietypowanych listach (*S-expressions*), co umożliwia dużą elastyczność, ale eliminuje sprawdzanie typów w czasie kompilacji. Z kolei makra w Scali 3 operują na typowanych drzewach składni (TASTy), zapewniając pełne bezpieczeństwo typów.

2.3.2. Template Haskell

Template Haskell [36] wprowadza programowanie wieloetapowe do języka Haskell poprzez cytaty i wstawki, podobnie jak Scala 3. Obie implementacje wykorzystują cytaty (`[ ... ]` w Haskell, `'{ ... }` w Scali) oraz wstawki (`$(...)` w Haskell, `${...}` w Scali). Główna różnica polega na tym, że Template Haskell wymaga specjalnego trybu kompilacji (`-XTemplateHaskell`), podczas gdy makra Scali 3 są standardową częścią języka. Dodatkowo Scala 3 oferuje bogatsze API refleksji (TASTy).

2.3.3. Makra w Rust

Język Rust oferuje dwa systemy makr: makra deklaratywne (`macro\_rules!`) oraz makra proceduralne [37, 38]. W porównaniu do Scali 3, makra proceduralne w Rust operują na tokenach (ang. *token stream*), co daje dużą kontrolę, ale utrudnia analizę semantyczną.

Z kolei makra w Scali 3 operują na typowanych AST, co umożliwia analizę semantyczną i sprawdzanie typów wygenerowanego kodu.

2.4. Zastosowania metaprogramowania w projekcie

System metaprogramowania Scali 3 stanowi fundament implementacji projektu *ALPACA*. Kluczowe zastosowania obejmują następujące techniki:

- generację klas anonimowych (sekcja 3.1.7) — wykorzystanie `Symbol.newClass` do programatycznego tworzenia typów w czasie kompilacji,
- transformację AST (sekcja 3.1.4) — przepisywanie właścicieli symboli (*re-owning*) poprzez `ReplaceRefs`,
- typy rafinowane (sekcja 3.1.8) — dynamiczne rozszerzanie typów o pola strukturalne poprzez `Refinement`,
- walidację w czasie kompilacji (sekcja 3.1.11) — wykrywanie błędów gramatyki przed wykonaniem programu.

Szczegółowa analiza implementacji tych mechanizmów zostanie przedstawiona w rozdziale 3.

2.5. Podsumowanie rozdziału

Rozdział przedstawił system metaprogramowania Scali 3 jako fundament teoretyczny dla projektu *ALPACA*. Kluczowe wnioski są następujące.

- System cytatów i wstawek (ang. *quotes and splices*) umożliwia bezpieczne przenoszenie kodu między fazami kompilacji.
- Bezpieczeństwo międzyetapowe (ang. *cross-stage safety*) zapobiega błędom związanym z dostępem do zmiennych z niewłaściwych faz.
- Refleksja TASTy dostarcza bogatego API do analizy i transformacji kodu w czasie kompilacji.
- Scala 3 łączy zalety systemów metaprogramowania z Lisp, Template Haskell i Rust, wprowadzając własne innowacje w zakresie bezpieczeństwa typów.

Mechanizmy te stanowią podstawę implementacji opisanej w rozdziale 3, gdzie zostaną zastosowane do konstrukcji lekserów i parserów w czasie kompilacji.

Rozdział 3

Implementacja

3.1. Praktyczna implementacja analizatora leksykalnego z wykorzystaniem makr w Scali 3

3.1.1. Wprowadzenie do studium przypadku

Rozdział przedstawia implementację systemu analizy leksykalnej wykorzystującego mechanizmy metaprogramowania Scali 3 [32]. Implementacja stanowi studium przypadku zastosowania technik opisanych w rozdziale drugim w kontekście automatycznej generacji analizatora leksykalnego. System transformuje deklaratywne reguły tokenizacji, wyrażone w języku dziedzinowym (DSL), w kod proceduralny wykonywany w czasie kompilacji, wykorzystując refleksję TASTy [28] oraz typy rafinowane [39].

System `alpaca.lexer` implementuje transformację deklaratywnych reguł tokenizacji zapisanych jako funkcja częściowa (ang. *partial function*) w kod proceduralny wykonywany w czasie kompilacji. Celem tej transformacji jest wyeliminowanie narzutu analizy wyrażeń regularnych i budowy automatów w czasie działania aplikacji, a także zapewnienie bezpieczeństwa typów dla wygenerowanych tokenów. Wykorzystuje przy tym pełne spektrum możliwości refleksji TASTy [28], włączając generację klas w czasie kompilacji, transformację drzew AST [34] oraz wyspecjalizowane typy refinement.

3.1.2. Interfejs użytkownika

System udostępnia interfejs języka dziedzinowego (DSL) oparty na dopasowaniu wzorców, umożliwiający deklaratywne wyrażenie reguł tokenizacji.

```
1 type LexerDefinition[Ctx <: LexerCtx] = PartialFunction[String, Token[?,
    Ctx, ?]]
```

Listing 3.1. Definicja typu LexerDefinition.

Definicja `LexerDefinition` reprezentuje reguły leksera jako funkcję częściową mapującą wzorce wyrażeń regularnych (jako ciągi znaków) na definicje tokenów. Wykorzystanie funkcji częściowej pozwala na naturalne wyrażenie reguł leksykalnych w idiomatycznej składni Scali.

Metoda `lexer` definiuje główny interfejs systemu.

```
transparent inline def lexer[Ctx <: LexerCtx](
  using Ctx withDefault LexerCtx.Default,
)(
  inline rules: Ctx ?=> LexerDefinition[Ctx],
)(using
  copy: Copyable[Ctx],
  betweenStages: BetweenStages[Ctx],
)(using inline
  debugSettings: DebugSettings,
): Tokenization[Ctx]
```

Listing 3.2. Punkt wejścia: transparent inline def lexer.

Modyfikator `transparent inline` zapewnia, że zwracany typ będzie dokładnie odpowiadał wygenerowanej strukturze, włączając typy refinement dla poszczególnych tokenów. Użycie parametrów kontekstowych (`using`) realizuje wzorzec dependency injection na poziomie systemu typów.

3.1.3. Implementacja makra

Makro przyjmuje wyrażenie reprezentujące reguły analizatora leksykalnego jako `Expr[Ctx ?=> LexerDefinition[Ctx]]` oraz instancje kontekstualnych klas pomocniczych. Parametr `using Quotes` dostarcza dostępu do API refleksji TASTy [23, 29, 32].

3.1.4. Analiza i transformacja drzewa składni

Dekonstrukcja funkcji częściowej

Kluczowym krokiem implementacji jest ekstrakcja reguł z definicji funkcji częściowej.

```
val Lambda(oldCtx :: Nil, Lambda(_, Match(_, cases: List[CaseDef]))) =
    rules.asTerm.underlying
```

Listing 3.3. Dekonstrukcja funkcji częściowej (dopasowanie AST do CaseDef).

Fragment 3.3 wykorzystuje dopasowanie wzorców w cytatach (ang. *quotes*) do dekonstrukcji [29] typowanego AST funkcji częściowej. Struktura `Lambda(\_, Match(\_, cases))` odpowiada wewnętrznej reprezentacji funkcji częściowej, gdzie `Match` zawiera listę przypadków `CaseDef`.

Transformacja i adaptacja referencji

Kluczową techniką jest zastąpienie referencji do starego kontekstu nowymi referencjami za pomocą klasy `ReplaceRefs`.

```
def replaceWithNewCtx(newCtx: Term) = new ReplaceRefs[quotes.type].apply(
  (find = oldCtx.symbol, replace = newCtx),
  (find = tree.symbol, replace = Select.unique(newCtx, "lastRawMatched")),
)
```

Listing 3.4. Zastąpienie referencji starego kontekstu nowymi (ReplaceRefs).

Transformacja realizuje proces przepisania właściciela (*re-owning*) symboli w AST, polegający na modyfikacji referencji kontekstowych w celu dostosowania ich do nowego zakresu leksykalnego [34]. Klasa `ReplaceRefs` udostępnia `TreeMap`, który podczas przejścia po AST podmienia referencje do wskazanych symboli na podane termy[34].

3.1.5. Ekstrakcja i kompilacja wzorców

Funkcja extractSimple

Funkcja `extractSimple` implementuje logikę dopasowania różnych typów definicji tokenów.

```
def extractSimple(
  ctxManipulation: Expr[CtxManipulation[Ctx]],
): PartialFunction[Expr[ThisToken], List[Expr[ThisToken]]] =
  case '{ Token.Ignored(using $ctx) } =>
  // ...

  case '{ type t <: ValidName; Token.apply[t](using $ctx) } =>
  // ...

  case '{ type t <: ValidName; Token.apply[t]($value: String)(using $ctx)
   } if value.asTerm.symbol == tree.symbol =>
  // ...

  case '{ type t <: ValidName; Token.apply[t]($value: v)(using $ctx) } =>
  // ...
```

Listing 3.5. Funkcja extractSimple: dopasowywanie definicji tokenów.

Wykorzystuje ona dopasowanie wzorców w cytatach (ang. *quotes*) z ekstraktorem typów[29], umożliwiając rozróżnienie różnych wariantów definicji tokenów na poziomie typów. Konstrukcja `type t <: ValidName` w wzorcu wiąże parametr typu do zmiennej wzorca `t`, umożliwiając jego późniejsze wykorzystanie.

Ekstrakcja definicji tokenów wymaga następnie ich analizy i walidacji, co realizuje klasa `CompileNameAndPattern`.

3.1.6. Analiza wzorców: klasa CompileNameAndPattern

Klasa `CompileNameAndPattern` odpowiada za ekstrakcję i walidację wzorców tokenów podczas ekspansji makra[32]. Jej głównym zadaniem jest transformacja wzorców występujących w definicjach DSL. Wzorce te są przekształcane w struktury `TokenInfo`, które następnie są wykorzystywane do generacji finalnego kodu leksera.

Implementacja wykorzystuje rekurencyjne przetwarzanie drzewa AST z zastosowaniem optymalizacji rekurencji ogonowej (`@tailrec`), co eliminuje ryzyko przepełnienia stosu dla złożonych wzorców.

3.1.7. Generacja klasy anonimowej

Kluczowym mechanizmem implementacyjnym makra `lexer` jest programatyczna konstrukcja klasy anonimowej w czasie kompilacji[27]. Proces ten wykorzystuje API refleksji

TASTy[28] do dynamicznego tworzenia struktur typów, które następnie są materializowane jako kod bajtowy JVM.

Anonimowa klasa implementująca `Tokenization[Ctx]` jest tworzona poprzez wywołanie `Symbol.newClass`.
Metoda `Symbol.newClass` przyjmuje następujące parametry:

- `Symbol.spliceOwner` — właściciel nowego symbolu w hierarchii definiowania, zapewniający poprawną widoczność w zakresie leksykalnym
- `Symbol.freshName(``\$anon'')` — generowanie unikalnej nazwy klasy zgodnie z konwencją kompilatora Scali dla klas anonimowych
- `List(TypeRepr.of[Tokenization[Ctx]])` — lista typów bazowych, w tym przypadku pojedyncza implementacja abstrakcyjnej klasy `Tokenization`
- `decls` — funkcja dostarczająca listy deklaracji członków klasy (pól i metod)

Funkcja `decls` konstruuje pełną listę deklaracji dla klasy anonimowej:

- dla każdego zdefiniowanego tokena tworzony jest symbol pola typu `DefinedToken[Name, Ctx, Value]`.
- `Type alias Fields` — typ pomocniczy w formie `NamedTuple` ułatwiający strukturalny dostęp do tokenów.
- `Pole compiled` — wartość typu `Regex` zawierająca skompilowane wyrażenie regularne dla wszystkich tokenów.
- `Pole tokens` — lista wszystkich zdefiniowanych tokenów (włączając ignorowane).
- `Pole byName` — czyli mapa umożliwiająca dynamiczny dostęp do tokenów po nazwie.

Po zdefiniowaniu symbolu klasy następuje konstrukcja jej ciała. Klasa jest następnie instancjonowana poprzez wywołanie jej konstruktora.

3.1.8. Typy rafinowane (refinement types)

Typy rafinowane (*refinement types*) stanowią mechanizm systemu typów Scali umożliwiający dodanie informacji o strukturze typu w czasie kompilacji [39]. W kontekście implementacji leksera typy rafinowane pozwalają na dodanie informacji o polach tokenów bezpośrednio do typu zwracanego przez makro.

Proces rafinowania typu

Typ wynikowy jest konstruowany poprzez iteracyjne rafinowanie typu bazowego[34].

```
definedTokens
  .unsafeFoldLeft(TypeRepr.of[Tokenization[Ctx]]):
    case (tpe, '{ $token: DefinedToken[name, Ctx, value] }) =>
      Refinement(tpe, ValidName.from[name], token.asTerm.tpe)
  .asType match
  case '[refinedTpe] =>
    val newCls =
    Typed(New(TypeIdent(cls)).select(cls.primaryConstructor).appliedToNone,
    TypeTree.of[refinedTpe])
     Block(clsDef :: Nil, newCls).asExprOf[Tokenization[Ctx] & refinedTpe]
```

Listing 3.6. Rafinowanie typu wynikowego o pola tokenów.

Funkcja `Refinement(tpe, name, memberType)` tworzy nowy typ będący rozszerzeniem typu. Operacja ta jest wykonywana w czasie kompilacji i nie generuje dodatkowego kodu w czasie wykonania.

Wynikowy typ

Wynikowy typ ma formę typu przecięcia (ang. *intersection type*).

```
Tokenization[Ctx] & {
  val TOKEN1: DefinedToken["NAME1", Ctx, Type1]
  val TOKEN2: DefinedToken["NAME2", Ctx, Type2]
  ...
}
```

Listing 3.7. Wynikowy typ leksera.

Ten typ reprezentuje wartości będące jednocześnie instancjami `Tokenization[Ctx]` oraz posiadające określone pola strukturalne (ang. *computed field names*).

Dostęp do pól tokenów odbywa się poprzez `trait Selectable`. Standardowa implementacja tego mechanizmu, opisana w dokumentacji [39], wprowadza narzut związany z dynamicznym wyborem nazwy pola (refleksja). W prezentowanym rozwiązaniu narzut ten jest eliminowany poprzez precyzyjne typowanie strukturalne. Aby mechanizm `Selectable` działał poprawnie ze strukturalnymi typami i nie wymagał refleksji, klasa generowana przez makro musi implementować `type Fields <: NamedTuple.AnyNamedTuple`[40].

W naszym podejściu makro generuje definicję `type Fields` zawierającą wszystkie zdefiniowane tokeny i ich typy, dzięki czemu udaje się uzyskać następujące rezultaty:

- IDE i kompilator dysponują informacją o dostępnych polach i ich typach (pełne uzupełnianie i sprawdzanie typów),
- wywołanie `c.NAZWA` jest bezpieczne typowo mimo mechanizmu dynamicznego wyboru nazwy.

```
val fieldTpe = definedTokens
  .unsafeFoldLeft[(Type[? <: Tuple], Type[? <:
   Tuple])]((Type.of[EmptyTuple], Type.of[EmptyTuple])):
    case (
          ('[type names <: Tuple; names], '[type types <: Tuple; types]),
          '{ $token: DefinedToken[name, Ctx, value] },
        ) =>
      (Type.of[name *: names], Type.of[Token[name, Ctx, value] *: types])
  .runtimeChecked
  .match
    case ('[type names <: Tuple; names], '[type types <: Tuple; types]) =>
   TypeRepr.of[NamedTuple[names, types]]
```

Listing 3.8. Tworzenie typuFields.

3.1.9. Uzasadnienie wybranego podejścia implementacyjnego

Eliminacja narzutu wykonania w czasie działania programu

Wszystkie definicje tokenów są rozwiązywane statycznie w czasie kompilacji[23]. Dostęp do tokenów realizowany jest jako bezpośrednie odwołanie do pola klasy, które w kodzie bajtowym JVM [41] reprezentowane jest przez instrukcję `getfield` o złożoności czasowej O(1). Teoretycznie eliminuje to narzut związany z operacjami dynamicznymi, choć pełna weryfikacja empiryczna tego założenia wykracza poza zakres niniejszej pracy.
Alternatywne podejście oparte na strukturze mapy wymagałoby:

1. obliczenia funkcji haszującej dla klucza,
2. przeszukiwania tablicy haszującej,
3. potencjalnej obsługi kolizji,
4. dynamicznego rzutowania typu.

Wprowadzałoby ono znaczący narzut wydajnościowy oraz eliminowało możliwość optymalizacji przez kompilator.

Bezpieczeństwo typów na poziomie systemu

Dzięki typom rafinowanym każdy token posiada precyzyjny typ znany kompilatorowi[39]. System typów weryfikuje poprawność wszystkich operacji w czasie kompilacji, eliminując możliwość błędów związanych z niepoprawnym typowaniem wartości tokenów.

Integracja z narzędziami deweloperskimi

Ponieważ tokeny są reprezentowane jako rzeczywiste pola w typie, środowiska deweloperskie (IDE) mogą wykorzystać informacje typu do realizacji następujących funkcjonalności.

- Automatycznego uzupełniania nazw tokenów
- Prezentacji pełnych sygnatur typów przy najechaniu kursorem
- Nawigacji do definicji przez mechanizm *go-to-definition*
- Wykrywania błędów składniowych przed kompilacją

Te funkcjonalności są niemożliwe do realizacji w przypadku dostępu przez struktury dynamiczne.

Statyczna detekcja konfliktów wzorców

Makro przeprowadza analizę wszystkich wzorców w czasie kompilacji, wykrywając potencjalne konflikty nakładających się wyrażeń regularnych. Mechanizm ten zapewnia, że błędy konfiguracji są wykrywane na etapie kompilacji, a nie w czasie wykonania programu, co jest zgodne z zasadą *fail-fast* w inżynierii oprogramowania.

Typowanie strukturalne z gwarancjami nominalnymi

Zastosowanie typów rafinowanych[39] łączy zalety typowania strukturalnego (elastyczność w dostępie do składowych) z bezpieczeństwem typowania nominalnego (jednoznaczna identyfikacja typów). Każde pole w typie rafinowanym ma precyzyjny typ nominalny,

podczas gdy dostęp do tych pól odbywa się przez nazwę, co zapewnia elastyczność interfejsu.

3.1.10. Analiza alternatywnych rozwiązań

Podejście oparte na mapowaniu dynamicznym

Alternatywne podejście mogłoby wykorzystywać strukturę mapującą do przechowywania tokenów.

```
class SimpleLexer {
  val tokens: Map[String, Token[?, ?, ?]] = Map(
    "NUMBER" -> ...,
    "PLUS" -> ...
  )
  def apply(name: String): Token[?, ?, ?] = tokens(name)
}
```

Listing 3.9. Podejście oparte na mapowaniu dynamicznym.

Do wad tego podejścia należą następujące aspekty.

- Brak bezpieczeństwa typów: błędne nazwy tokenów wykrywane są dopiero w czasie wykonania
- Utrata informacji o typach: zwracany typ to egzystencjalny `Token[?, ?, ?]`
- Narzut wydajnościowy operacji haszowania i przeszukiwania
- Brak wsparcia narzędzi deweloperskich

Podejście oparte na jawnej definicji klasy

Innym rozwiązaniem byłoby jawne definiowanie klasy leksera przez użytkownika.

```
class MyLexer extends Tokenization[DefaultGlobalCtx]:
  val NUMBER = DefinedToken[...]
  val PLUS = DefinedToken[...]
  protected def compiled: Regex = "(?<token0>[0-9]+)|(?<token1>\\+)".r
  // ...
```

Listing 3.10. Podejście oparte na jawnej definicji klasy.

Główne wady tego podejścia są następujące.

- Wysoki poziom redundancji kodu (*boilerplate*)
- Konieczność ręcznej kompilacji wyrażeń regularnych
- Podatność na błędy synchronizacji między definicjami tokenów a wyrażeniem regularnym
- Brak mechanizmu DSL ułatwiającego definicję reguł

3.1.11. Walidacja i obsługa błędów

Walidacja wzorców regularnych

System wykorzystuje pomocniczą klasę `RegexChecker` do walidacji wzorców. Mechanizm ten sprawdza poprawność składni wyrażeń regularnych już w czasie kompilacji i ra-

portuje błędy z dokładną lokalizacją wzorca. Metoda `report.errorAndAbort` przerywa kompilację i wyświetla komunikat o błędzie, eliminując konieczność detekcji błędów w czasie wykonania, co jest zgodne z zasadą wczesnej walidacji (*fail-fast*) [32, 33].

Obsługa nieobsługiwanych konstrukcji

Kod jawnie sygnalizuje nieobsługiwane przypadki. Obsługiwane są wyłącznie jasno zdefiniowane formy wzorców; w przypadku napotkania innej konstrukcji kompilacja jest przerywana z komunikatem zawierającym szczegóły AST, co upraszcza diagnostykę i utrzymuje zasadę fail-fast. Ta strategia jest zgodna z zasadą fail-fast - lepiej jest wyraźnie odrzucić nieobsługiwane konstrukcje niż milcząco generować niepoprawny kod.

3.2. Praktyczna implementacja generatora parserów z wykorzystaniem makr w Scali 3

3.2.1. Wprowadzenie do generatora parserów

Implementacja generatora parserów w systemie *ALPACA* wykorzystuje mechanizmy metaprogramowania Scali 3[32] do konstrukcji tabel parsowania LR(1) w czasie kompilacji. Podejście to łączy zalety generatorów kodu (wydajność wykonania, statyczna walidacja gramatyki) z elastycznością bibliotek (integracja z systemem typów, brak dodatkowego kroku kompilacji).
Realizacja napotkała szereg wyzwań technicznych, z których najistotniejsze to:

- konstrukcja tabel LR(1) w czasie kompilacji z wykorzystaniem makr,
- integracja z systemem typów Scali w akcjach semantycznych,
- obejście ograniczenia rozmiaru metod JVM poprzez fragmentację generowanego kodu,
- deklaratywny mechanizm rozwiązywania konfliktów gramatycznych,
- walidacja gramatyk podczas kompilacji z komunikatami o błędach.

W szczególności ograniczenie rozmiaru metod JVM ilustruje istotny aspekt praktycznego metaprogramowania: generowany kod musi nie tylko być poprawny funkcjonalnie, ale również respektować wszystkie techniczne ograniczenia platformy docelowej.

3.2.2. Interfejs API parsera

Definicja parsera

Użytkownik definiuje parser poprzez dziedziczenie po klasie bazowej `Parser[Ctx]`.

```
abstract class Parser[Ctx <: ParserCtx](
  using Ctx withDefault ParserCtx.Empty,
)(using
  empty: Empty[Ctx],
  tables: Tables[Ctx],
)
```

Listing 3.11. Klasa bazowa Parser.

Typ parametryczny `Ctx` reprezentuje globalny kontekst parsera, umożliwiający przechowywanie stanu między akcjami semantycznymi (np. tablicę symboli). Parametr kontekstualny `tables: Tables[Ctx]` jest automatycznie generowany przez makro i zawiera tabele parsowania oraz akcji semantycznych.

Definicja reguł gramatycznych

Reguły gramatyczne definiowane są jako wartości typu `Rule[R]`, gdzie `R` określa typ wyniku redukcji.

```
    { case Assignment(a) => a },
  )

  def Statement: Rule[AST.Statement] = rule(
    { case ML.break(l) => AST.Break },
```

Listing 3.12. Przykład definicji reguł parsera.

Składnia wykorzystuje dopasowanie wzorców Scali do wyrażenia produkcji gramatycznych. Każdy przypadek (`case`) reprezentuje pojedynczą produkcję, gdzie lewa strona wzorca odpowiada prawej stronie produkcji gramatycznej, a wyrażenie po strzałce (`=>`) definiuje akcję semantyczną. Na przykład wzorzec `\{ case (Expr(a), CalcLexer.PLUS(\_), Expr(b)) => a + b \}` odpowiada produkcji `Expr → Expr PLUS Expr` z akcją sumującą wartości podwyrażeń.

3.2.3. Generacja tabel parsowania w czasie kompilacji

Centralnym elementem systemu jest makro `createTablesImpl`, które analizuje definicję parsera i generuje tabele w czasie kompilacji. Wykonuje ono następujące kroki.

1. ekstrakcję wszystkich reguł gramatycznych z definicji parsera poprzez refleksję TASTy,
2. transformację wzorców dopasowania na produkcje gramatyczne,
3. konstrukcję automatów LR(1) i tabel parsowania,
4. generację tabel akcji semantycznych,
5. walidację gramatyki i rozwiązywanie konfliktów.

Funkcja `extractEBNF` dokonuje transformacji wzorców dopasowania na produkcje gramatyczne.

Kluczowym wyzwaniem jest zachowanie poprawności referencji do symboli przy przenoszeniu kodu akcji semantycznej z kontekstu definicji reguły do wygenerowanej tabeli akcji. Wymaga to zastosowania techniki *re-owning* symboli, realizowanej przez klasę `ReplaceRefs`.

3.2.4. Trudne problemy rozwiązane w implementacji

Problem ograniczenia rozmiaru metod JVM

Jednym z kluczowych wyzwań technicznych napotkanych podczas implementacji było ograniczenie rozmiaru metod w maszynach wirtualnych JVM. Zgodnie ze specyfikacją JVM[41], rozmiar kodu bajtowego pojedynczej metody nie może przekroczyć 65536 bajtów

(64 KB). Dla złożonych gramatyk z dużą liczbą stanów i produkcji, wygenerowane tabele parsowania mogą zawierać tysiące wpisów, co przy naiwnej implementacji prowadziło do przekroczenia tego limitu.

Problem manifestował się podczas próby wyrażenia tabeli parsowania jako literału mapowego w kodzie.

```
'{
    Map(
      (0, Terminal("PLUS")) -> Shift(1),
      (0, Terminal("NUMBER")) -> Shift(2),
      ...
      (999, Terminal("EOF")) -> Reduction(prod),
    )
  }
```

Listing 3.13. Naiwna implementacja prowadząca do przekroczenia limitu.

Takie podejście generuje pojedynczą, dużą metodę zawierającą wszystkie wpisy tabeli, co dla gramatyk o rozmiarze produkcyjnym skutkuje błędem kompilacji `Method too large`[42].

Problem został rozwiązany poprzez zastosowanie techniki *fragmentacji metod*. Zamiast generować jeden duży literal mapy, każdy wpis tabeli jest dodawany w osobnej, małej metodzie pomocniczej.

```
      val additions = entries
        .map(entry =>
          '{
            def avoidTooLargeMethod(): Unit = $builder += ${ Expr(entry) }
            avoidTooLargeMethod()
          }.asTerm,
        )
        .toList
```

Listing 3.14. Rozwiązanie problemu rozmiaru metod przez fragmentację.

W tym podejściu:

1. Tworzymy builder mapy jako zmienną lokalną (`Map.newBuilder`).
2. Każde dodanie wpisu do buildera jest opakowane w osobną metodę `avoidTooLargeMethod()`.
3. Metody te są wywoływane sekwencyjnie jako lista wyrażeń w bloku.
4. Końcowy wynik jest uzyskiwany przez wywołanie `builder.result()`.

Zastosowana technika fragmentacji skutecznie eliminuje problem przekroczenia limitu rozmiaru metody. Każda metoda pomocnicza zawiera jedynie kilka instrukcji bajtowych (typowo 5–10 w zależności od złożoności wpisu tabeli), co gwarantuje zgodność ze specyfikacją JVM [41]. Dodatkowo kompilator JIT może efektywnie zoptymalizować te metody poprzez *inlining*, eliminując narzut wywołań funkcji w czasie wykonania.

Rozwiązanie to ilustruje ważną lekcję w metaprogramowaniu: kod generowany przez makra musi respektować wszystkie ograniczenia platformy docelowej, które normalnie są niewidoczne dla programistów piszących kod ręcznie.

Zachowanie bezpieczeństwa typów w akcjach semantycznych

Kolejnym istotnym wyzwaniem jest zapewnienie bezpieczeństwa typów w akcjach semantycznych podczas transformacji kodu z kontekstu makra do wygenerowanych tabel. Akcje semantyczne definiowane przez użytkownika mogą odwoływać się do następujących elementów.

- kontekstu parsera (`ctx`),
- wartości z dopasowanych symboli gramatycznych,
- zewnętrznych funkcji i wartości.

Problem polega na tym, że te referencje muszą zostać przepisane podczas przenoszenia kodu akcji z miejsca definicji do tabeli akcji. Funkcja `createAction` realizuje tę transformację.

```
1     def createAction(binds: List[Option[Bind]], rhs: Term) =
    createLambda[Action[Ctx]]:
2       case (methSym, (ctx: Term) :: (param: Term) :: Nil) =>
3         val seqApplyMethod =
    param.select(TypeRepr.of[Seq[Any]].typeSymbol.methodMember("apply").head)
4         val seq = param.asExprOf[Seq[Any]]
5
6         val replacements = (find = ctxSymbol, replace = ctx) ::
7           binds.zipWithIndex
8             .collect:
9               case (Some(bind), idx) => ((bind.symbol,
    bind.symbol.typeRef.asType), Expr(idx))
10            .unsafeFlatMap:
11              case ((bind, '[t]), idx) => Some((find = bind, replace =
    '{ $seq($idx).asInstanceOf[t] }.asTerm))
12
13        replaceRefs(replacements*).transformTerm(rhs)(methSym)
```

Listing 3.15. Tworzenie akcji semantycznej z zachowaniem referencji.

Kluczowe aspekty implementacji obejmują następujące punkty.

1. Akcja jest transformowana w funkcję przyjmującą kontekst (`ctx`) oraz listę dzieci w drzewie parsowania (`param`).
2. Referencje do kontekstu parsera są zastępowane parametrem funkcji.
3. Wartości z dopasowanych symboli są ekstrahowane z listy dzieci poprzez indeksowanie.

System typów zapewnia, że ekstrakcje są bezpieczne względem typów dzięki informacji z wzorca dopasowania

Rozwiązywanie konfliktów gramatycznych

Parser LR może napotkać konflikty typu shift–reduce lub reduce–reduce podczas konstrukcji tabel parsowania. System ALPACA oferuje deklaratywny mechanizm rozwiązywania takich konfliktów poprzez relacje precedencji.

```
1 override val resolutions = Set(production.times.before(Lexer.PLUS),
    production.plus.after(Lexer.TIMES))
```

Listing 3.16. Deklaracja rozwiązań konfliktów.

Implementacja wykorzystuje klasę `ConflictResolutionTable`, która podczas konstrukcji tabeli parsowania wykonuje następujące czynności.

1. Wykrywa konflikty między akcjami dla danego stanu i symbolu
2. Analizuje zdefiniowane przez użytkownika relacje precedencji
3. Wybiera odpowiednią akcję zgodnie z deklaracją
4. Zgłasza błąd kompilacji dla nierozwiązanych konfliktów

To podejście umożliwia wyrażenie precedencji i łączności operatorów w sposób bardziej naturalny niż tradycyjne narzędzia `\%left`, `\%right` i `\%nonassoc` w SLY[11].

3.2.5. Generacja kodu tabel

System wymaga konwersji struktur danych w czasie kompilacji (wartości) na kod (wyrażenia `Expr[T]`). Realizowane jest to poprzez implementację instancji `ToExpr` dla typów `ParseTable` i `ActionTable`.

Implementacja `ToExpr[ParseTable]` jest szczególnie interesująca, gdyż musi radzić sobie z potencjalnie dużymi tabelami (patrz 3.2.4).

```
given ToExpr[ParseTable] with
  def apply(entries: ParseTable)(using quotes: Quotes): Expr[ParseTable] =
    import quotes.reflect.*

    type BuilderTpe = mutable.Builder[
      ((state: Int, stepSymbol: parser.Symbol), Shift | Reduction),
      Map[(state: Int, stepSymbol: parser.Symbol), Shift | Reduction],
    ]

    val symbol = Symbol.newVal(
      Symbol.spliceOwner,
      Symbol.freshName("builder"),
      TypeRepr.of[BuilderTpe],
      Flags.Mutable,
      Symbol.noSymbol,
    )

    val valDef = ValDef(symbol, Some('{ Map.newBuilder: BuilderTpe }.asTerm))

    val builder = Ref(symbol).asExprOf[BuilderTpe]

    val additions = entries
      .map(entry =>
        '{
          def avoidTooLargeMethod(): Unit = $builder += ${ Expr(entry) }
          avoidTooLargeMethod()
        }.asTerm,
      )
      .toList

    val result = '{ $builder.result() }.asTerm

    Block(valDef :: additions, result).asExprOf[ParseTable]
```

Listing 3.17. Implementacja ToExpr dla ParseTable.

Ta implementacja demonstruje zaawansowane techniki metaprogramowania.

1. Tworzenie nowych symboli (`Symbol.newVal`) reprezentujących zmienne w generowanym kodzie.
2. Konstrukcja definicji wartości (`ValDef`) z przypisaniem początkowym.
3. Generacja listy wyrażeń manipulujących builderem.
4. Składanie wszystkiego w blok kodu (`Block`) z finalnym wynikiem.

3.3. Narzędzia pomocnicze

Implementacja systemu *ALPACA* wykorzystuje zaawansowane mechanizmy metaprogramowania Scali 3, w tym refleksję TASTy [28], derywację typów [43] oraz transformację drzew składni abstrakcyjnej (AST) [34]. Realizacja tych mechanizmów wymaga zestawu narzędzi pomocniczych abstrahujących typowe wzorce operacji na typach i drzewach. Niniejsza sekcja przedstawia cztery kluczowe komponenty infrastrukturalne:

- `Empty[T]` — generyczna konstrukcja wartości domyślnych dla typów produktowych,
- `ReplaceRefs` — transformacja drzew AST poprzez podstawianie symboli,
- `CreateLambda` — programatyczna konstrukcja wyrażeń funkcyjnych w czasie kompilacji,
- `Copyable[T]` — generyczna funkcja kopiowania dla klas przypadku (*case classes*).

Narzędzia te realizują wzorce projektowe typowe dla systemów opartych na makrach kompilacyjnych [44], eliminując powtarzalny kod (*boilerplate*) oraz zapewniając bezpieczeństwo typów na poziomie kompilacji.

3.3.1. Empty[T] — konstrukcja wartości domyślnych

Klasa typu `Empty[T]` stanowi abstrakcję nad mechanizmem konstrukcji wartości domyślnych dla typów produktowych (ang. *product types*) [45]. W systemie typów Scali [46] typy produktowe odpowiadają klasom przypadku (*case classes*) oraz krotkom (*tuples*), będącym reprezentacją iloczynów kartezjańskich typów składowych.

Podczas ekspansji makr kompilacyjnych często zachodzi potrzeba utworzenia instancji typu `T` bez znajomości jego konkretnej struktury. Standardowe podejście wymagałoby podjęcia następujących działań.

- ręcznej specyfikacji wartości wszystkich pól,
- naruszenia abstrakcji poprzez dostęp do wewnętrznej struktury typu,
- utraty bezpieczeństwa typów w przypadku zmiany definicji `T`.

Klasa typu `Empty[T]` rozwiązuje ten problem poprzez automatyczną derywację funkcji konstruującej na podstawie wartości domyślnych parametrów konstruktora [43].

Typ `Empty[T]` jest reprezentowany jako funkcja zerargumentowa `() => T`, co umożliwia leniwą inicjalizację instancji (*lazy instantiation*). Atrybut `private[alpaca]` ogranicza widoczność do pakietu, zapobiegając przypadkowemu użyciu poza systemem.

```
trait Empty[T] extends (() => T)
```

Listing 3.18. Definicja klasy typu Empty[T].

Derywacja instancji `Empty[T]` wykorzystuje mechanizm `Mirror` wprowadzony w Scali 3 [43], który umożliwia generyczną introspekcję typów produktowych w czasie kompilacji. Kompilator automatycznie generuje kod konstruujący instancję `T` z wartości domyślnych, weryfikując przy tym ich dostępność.

```
case class Config(
  name: String = "default",
  count: Int = 0,
)
// compiler automatically derives Empty[Config]
val empty = summon[Empty[Config]]
val instance: Config = empty() // Config("default", 0)
```

Listing 3.19. Użycie mechanizmu Empty[T].

Bez mechanizmu `Empty[T]` konstrukcja wartości domyślnej w kontekście generycznym wymagałaby od użytkownika jawnego podania domyślnych wartości dla każdego typu `T`, co wyeliminowałoby zalety programowania generycznego. `Empty[T]` eliminuje te problemy poprzez derywację w czasie kompilacji, zachowując bezpieczeństwo typów oraz zerowy narzut wykonania.
Mechanizm derywacji wymaga, aby:

- typ `T` był typem produktowym (klasa przypadku lub krotka),
- wszystkie parametry konstruktora miały wartości domyślne,
- wartości domyślne były obliczalne w czasie kompilacji.

Naruszenie tych warunków prowadzi do błędu kompilacji z komunikatem wskazującym brakujące wartości domyślne.

3.3.2. ReplaceRefs — transformacja symboli w AST

Klasa `ReplaceRefs` rozszerza `TreeMap` — abstrakcyjną klasę bazową dla transformacji drzew składni abstrakcyjnej w systemie refleksji TASTy [34]. Klasa `TreeMap` definiuje wzorzec projektowy odwiedzającego (*visitor pattern*) [47] dla typowanego AST Scali 3, umożliwiając rekurencyjne przetwarzanie węzłów drzewa z zachowaniem bezpieczeństwa typów.

Podczas ekspansji makr kompilacyjnych często zachodzi potrzeba adaptacji fragmentów kodu z jednego kontekstu leksykalnego do innego. Przykładem jest sytuacja, w której kod oryginalnie odnoszący się do parametru makra `ctx` musi zostać przepisany tak, aby odnosił się do parametru metody w wygenerowanej klasie `newCtx`. Proces ten, znany jako *re-owning* [34], wymaga systematycznej zamiany wszystkich referencji do starego symbolu nowymi referencjami.

```
class ReplaceRefs[Q <: Quotes](using val quotes: Q)
```

Listing 3.20. Definicja klasy ReplaceRefs rozszerzającej TreeMap.

Klasa `ReplaceRefs` implementuje metodę `transformTree`, która:

1. przechodzi rekurencyjnie po wszystkich węzłach drzewa AST,
2. identyfikuje referencje do symboli wymienionych w mapie podstawień,
3. zastępuje te referencje odpowiednimi termami zastępczymi,
4. zachowuje strukturę typów oraz kontekst właściciela symbolu (*owner*).

Transformacja jest realizowana w sposób strukturalnie rekurencyjny, co gwarantuje kompletność zamiany oraz zachowanie poprawności typowania.

Fragment 3.21 ilustruje zastąpienie referencji do parametru makra `oldCtx` nowym symbolem `newCtx` w ciele wygenerowanej metody.

```
// during macro expansion:
val oldCtxSymbol: Symbol = ... // macro parameter symbol
val newCtxRef: Term = Ref(newCtxSymbol) // reference to new symbol

val replaceRefs = ReplaceRefs()
val treeMap = replaceRefs((oldCtxSymbol, newCtxRef))

// during function body transformation:
val originalBody: Term = ... // function body that references oldCtx
val transformedBody: Term = treeMap.transformTree(originalBody)(owner)
// all occurrences of oldCtxSymbol are now replaced with newCtxRef
```

Listing 3.21. Użycie ReplaceRefs w kontekście ekspansji makra.

W rezultacie kod oryginalnie odnoszący się do `oldCtx.field` jest transformowany do `newCtx.field`, co umożliwia prawidłowe działanie wygenerowanego kodu w nowym kontekście leksykalnym.

3.3.3. CreateLambda — programatyczna konstrukcja wyrażeń funkcyjnych

Klasa `CreateLambda` umożliwia programatyczną konstrukcję wyrażeń funkcyjnych (lambda) w czasie kompilacji. W kontekście makr kompilacyjnych bezpośrednie użycie składni lambda języka Scala jest niemożliwe, ponieważ makro operuje na reprezentacjach AST, a nie na kodzie źródłowym [33].

Podczas generacji kodu w makrach często zachodzi potrzeba utworzenia funkcji, której ciało jest konstruowane dynamicznie na podstawie analizy typów lub struktur danych dostępnych w czasie kompilacji. Przykładem jest generacja funkcji transformującej dla parsera, która ekstrahuje wartości z dopasowanych tokenów i konstruuje węzeł AST. Parametry takiej funkcji (symbole tokenów) są znane dopiero w momencie ekspansji makra, co uniemożliwia użycie statycznej składni lambda.

```
class CreateLambda[Q <: Quotes](using val quotes: Q)
```

Listing 3.22. Definicja klasy CreateLambda do programatycznej konstrukcji wyrażeń lambda.

Klasa `CreateLambda` implementuje algorytm konstrukcji wyrażeń lambda poprzez następujące kroki.

1. utworzenie świeżego symbolu dla każdego parametru funkcji,
2. wywołanie funkcji użytkownika dostarczającej ciała na podstawie tych symboli,
3. konstrukcję węzła `Lambda` w AST z odpowiednimi typami parametrów i zwracanym,
4. weryfikację typowania wyniku.

Mechanizm ten jest analogiczny do konstrukcji `Lambda` w systemie refleksji TASTy [34], ale oferuje wyższy poziom abstrakcji poprzez automatyczne zarządzanie symbolami i kontekstem właściciela.

```
1 val createLambda = CreateLambda()
2 val lambdaExpr: Expr[Int => Int] = createLambda[Int => Int] { case
    (methodSymbol, List(argTree)) =>
3 // building function body based on method symbol and arguments
4 }
```

Listing 3.23. Użycie CreateLambda do konstrukcji wyrażenia funkcyjnego.

3.3.4. Copyable[T] — generyczna funkcja kopiowania

Klasa typu `Copyable[T]` definiuje operację kopiowania (*shallow copy*) dla typów produktowych. W systemie *ALPACA* kopiowanie jest wykorzystywane do tworzenia nowych instancji kontekstu parsera z modyfikowanymi polami (np. aktualizacja numeru wiersza po napotkaniu znaku nowej linii), bez konieczności ręcznej rekonstrukcji całej struktury.

Klasy przypadku w Scali oferują automatycznie generowaną metodę `copy`, która umożliwia tworzenie zmodyfikowanych kopii instancji. Jednak w kontekście programowania generycznego, gdzie typ `T` jest parametrem, dostęp do metody `copy` wymaga refleksji strukturalnej [39], co wprowadza narzut wydajnościowy. Klasa typu `Copyable[T]` rozwiązuje ten problem poprzez derywację funkcji kopiującej w czasie kompilacji, eliminując narzut wykonania.

```
1 @implicitNotFound("${T} should be a case class.")
2 trait Copyable[T] extends (T => T)
```

Listing 3.24. Definicja klasy typu Copyable[T].

Anotacja `@implicitNotFound` dostarcza czytelny komunikat o błędzie w przypadku próby użycia `Copyable[T]` dla typu niebędącego klasą przypadku.

Derywacja instancji `Copyable[T]` wykorzystuje mechanizm `Mirror.ProductOf[T]`, który umożliwia dekonstrukcję instancji typu produktowego do krotki wartości pól, a następnie rekonstrukcję nowej instancji z tej krotki. Proces ten jest realizowany w czasie kompilacji bez użycia refleksji w czasie wykonania [43].

```scala
case class User(
  name: String,
  age: Int,
)

// compiler automatically derives Copyable[User]
// val copy = summon[Copyable[User]]
// val user = User("Alice", 30)
// val copied: User = copy(user) // User("Alice", 30)
```

Listing 3.25. Użycie Copyable[T].

Operacja kopiowania realizowana przez `Copyable[T]` jest kopią płytką (*shallow copy*) — pola będące referencjami do obiektów wskazują na te same instancje w oryginalnej i skopiowanej strukturze. Dla kontekstów parsera, które zawierają głównie typy wartościowe oraz niemodyfikowalne struktury, ograniczenie to nie stanowi problemu.

Analiza wydajności

Teoretycznie, operacja kopiowania realizowana przez `Copyable[T]` powinna być równoważna wywołaniu metody `copy`, ponieważ obie sprowadzają się do konstrukcji nowej instancji z tych samych wartości pól. W praktyce `Copyable[T]` eliminuje narzut związany z refleksją strukturalną, który występowałby przy dostępie do `copy` w kontekście generycznym.

Empiryczna weryfikacja tego założenia wykracza poza zakres niniejszej pracy. W kontekście systemu *ALPACA* korzyść z unifikacji interfejsu (klasa typu) przewyższa potencjalne różnice wydajnościowe, które są pomijalnie małe dla operacji kopiowania struktur o niewielkim rozmiarze (kilka pól).

3.3.5. Porównanie z istniejącymi bibliotekami

Ekosystem Scali oferuje biblioteki dedykowane programowaniu generycznemu, takie jak Shapeless [48] i Magnolia [49], które realizują derywację klas typów dla typów produktowych i sumarycznych. Wybór własnej implementacji narzędzi `Empty[T]` i `Copyable[T]` w systemie *ALPACA* wynikał z następujących przesłanek.

Minimalizacja zależności

Biblioteki takie jak Shapeless wprowadzają znaczące zależności oraz wydłużają czas kompilacji ze względu na złożone mechanizmy derywacji oparte na typach zależnych. Dla projektu o wąskim zakresie funkcjonalności koszt ten jest nieuzasadniony.

Wykorzystanie natywnych mechanizmów Scali 3

Scala 3 wprowadza mechanizm `Mirror` [43], który eliminuje potrzebę stosowania makr w stylu Shapeless, redukując złożoność implementacji. Własna implementacja oparta na `Mirror` jest prostsza, bardziej czytelna oraz lepiej wspierana przez narzędzia IDE niż rozwiązania oparte na starszych mechanizmach metaprogramowania.

Kontrola nad komunikatami błędów

Biblioteki generyczne generują często nieczytelne komunikaty błędów związane z wewnętrznymi abstrakcjami. Własna implementacja pozwala dostosować komunikaty błędów do kontekstu systemu *ALPACA*.

Rozdział 4

Algorytmy analizy leksykalnej

4.1. Teoretyczne podstawy

Analizator leksykalny (lekser) stanowi fundamentalną fazę przetwarzania tekstu źródłowego, przekształcając sekwencję znaków w ciąg jednostek leksykalnych (tokenów), które reprezentują niepodzielne elementy składniowe dla fazy analizy składniowej [50]. Klasyczna konstrukcja analizatora opiera się na połączeniu teorii formalnych języków oraz teorii automatów skończonych [51].

4.1.1. Opis języka tokenów

Każda klasa tokenów jest definiowana poprzez język regularny: zbiór słów akceptowanych przez wyrażenie regularne. Zbiór reguł tokenów stanowi sumę języków regularnych; ich unia jest również językiem regularnym [52], co umożliwia kompilację ich do jednolitego automatu deterministycznego.

4.1.2. Automaty skończone

Wyrażenia regularne są transformowane do postaci niedeterministycznej (NFA) poprzez konstrukcję Thompsona [53]. Następnie deterministyczna postać automatu (DFA) konstruowana jest poprzez algorytm usuwania niedeterminizmu (ang. *powerset construction*), polegający na iteracyjnym łączeniu zbiorów stanów NFA. Opcjonalnie przeprowadza się minimalizację automatu poprzez usuwanie stanów równoważnych [52].

DFA przetwarza wejście znak po znaku, zachowując jednoznaczny stan aktywny oraz informując, czy aktualny prefiks odpowiada jednemu z zdefiniowanych tokenów.

4.1.3. Strategia wyboru dopasowania

Lekser stosuje dwie komplementarne zasady determinujące zachowanie dla wieloznacznych sytuacji:

- Dopóki DFA ma ścieżkę przejść, znak jest konsumowany; token jest emitowany dopiero po ostatnim stanie akceptującym widzianym na tej ścieżce (najdłuższe dopasowanie, ang. *maximal munch*).
- Gdy kilka reguł akceptuje prefiks o tej samej długości, wybierana jest reguła o najwyższym priorytecie (często określanym kolejnością definicji).

Kombinacja tych zasad gwarantuje deterministyczną oraz reproducywalną sekwencję tokenów bez konieczności specjalnych mechanizmów rozstrzygania konfliktów.

4.1.4. Błędy leksykalne

W sytuacji, gdy automat nie posiada przejścia dla bieżącego znaku, zgłaszany jest błąd leksykalny w bieżącej pozycji wejścia. Mechanizm diagnostyczny przywołuje informacje o pozycji znaku, co znacząco ułatwia śledzenie źródła problemu w tekście źródłowym.

4.2. Automaty DFA a wyrażenia regularne

4.2.1. Tło: Tradycyjne podejście

Narzędzia klasyczne do budowy leksera (takie jak Lex [7]) generują jawny, deterministyczny automat skończony (DFA) z zestawu wyrażeń regularnych. Podejście to wymaga implementacji pełnego zestawu algorytmów: transformacja NFA→DFA, minimalizacja, optymalizacja — każdy etap jest czasochłonny dla twórcy narzędzia.

4.2.2. Alternatywa: Wyrażenia regularne biblioteczne

W systemie ALPACA podejście tradycyjne zostało zastąpione mechanizmem wykorzystującym natywny silnik wyrażeń regularnych biblioteki standardowej Scali [54]. Zamiast ręcznie kodować DFA, makro kompilacyjne łączy wszystkie wzorce operatorem alternatywy (`|`) w jedno wyrażenie regularne o nazwanych grupach, umożliwiając rozróżnienie, która reguła dopasowała się podczas każdego przebiegu.

4.2.3. Zalety podejścia opartego na wyrażeniach regularnych

Poprzez wykorzystanie powszechnie znanego mechanizmu wyrażeń regularnych, użytkownicy języka specjalistycznego (DSL) mogą definiować reguły leksykalne bez konieczności zrozumienia złożoności konstrukcji automatu i algorytmów optymalizacji. Użytkownicy mogą zastosować rozszerzenia wyrażeń regularnych (takie jak *backreference*, *negative lookahead*, czy warunkowość), których implementacja byłaby niemożliwa w jawnym DFA. Własna implementacja DFA wymaga pokrycia pełnego spektrum funkcjonalności wyrażeń regularnych, ciągłego utrzymania w synchronizacji z ewolucją języka hosta, oraz inwestycji w optymalizację. ALPACA deleguje ten wysiłek do zoptymalizowanego i wielokrotnie przetestowanego silnika bibliotecznego. Definicje reguł leksykalnych pozostają kompaktowe i czytelne, co ułatwia przegląd, weryfikację i modyfikację.

4.2.4. Wady i ograniczenia

Silniki wyrażeń regularnych mogą wykazywać wyższą złożoność obliczeniową niż ręcznie zoptymalizowane DFA, zwłaszcza w przypadku dużych zbiorów reguł lub złożonych wzorców. Zachowanie silnika wyrażeń regularnych dla dwuznacznych sytuacji (na przykład gdy dwa wzorce różnej długości akceptują identyczną sekwencję) zależy od implementacji silnika. W niekorzystnych przypadkach może prowadzić do nieoczekiwanych wyborów. Rozwiązaniem jest jawna deklaracja priorytetu poprzez kolejność definiowania reguł.

4.2.5. Decyzja

Rozważania przedstawione w sekcjach 4.2.3 i 4.2.4 doprowadziły do wyboru wyrażeń regularnych nad jawną konstrukcją DFA. Wymiana wydajności (potencjalnie) za uproszczenie interfejsu jest akceptowalna w kontekście systemu ALPACA.

4.3. Praktyczna implementacja leksera

Implementacja modułu `alpaca.lexer` łączy ergonomię języka specjalistycznego (DSL) z kodem wykonywany w czasie kompilacji, eliminując narzut parsowania wyrażeń regularnych w czasie działania aplikacji.

4.3.1. Przebieg tokenizacji

Podczas kompilacji projektu wszystkie wzorce są łączone operatorem alternatywy (`|`) w jedno wyrażenie regularne z nazwanymi grupami. Mechanizm ten umożliwia rozróżnienie, która z reguł dopasowała się podczas każdego przebiegu wyszukiwania. Następnie pętla skanująca — podczas wykonania aplikacji iteracyjnie wywołuje to wyrażenie na kolejnych fragmentach wejścia, identyfikuje dopasowany token, akumuluje leksemy oraz przesuwa wskaźnik wejścia aż do wyczerpania danych.

4.3.2. Obsługa reguł ignorowanych

System ALPACA umożliwia oznaczenie reguł jako „ignorowanych”. Takie reguły są wbudowywane we wspólny wzorzec, jednak ich dopasowania nie generują tokenów wyjściowych. Ujednolicony przebieg pętli skanującej upraszcza kod: ignorowane tokeny różnią się od normalnych wyłącznie akcją podejmowaną po dopasowaniu (brak emisji leksemu, zamiast tego aktualizacja stanu wewnętrznego).

4.3.3. Stanowa analiza leksykalna i rozszerzenia kontekstu

Możliwa jest stanowa analiza leksykalna poprzez utrzymywanie stanu maszyny stanów w obiekcie kontekstu. System ALPACA wewnętrznie definiuje mechanizm `BetweenStages`, który jest wywoływany po każdym rozpoznaniu leksemu i umożliwia modyfikację stanu kontekstu. Domyślna implementacja rejestruje ostatni leksem oraz śledzi numer linii i kolumny; użytkownik może jednak rozszerzyć tę logikę o własne zachowania, na przykład weryfikację poprawności zagnieżdżenia nawiasów.

4.3.4. Diagnostyka błędów leksykalnych

Gdy algorytm skanowania nie odnajduje prefiksu (brak przejścia w automacie), lekser zgłasza błąd leksykalny. Mechanizm `BetweenStages` umożliwia zbieranie danych kontekstowych (numer linii, kolumny, ostatni prawidłowy leksem), które następnie wzbogacają raport diagnostyczny. To podejście znacząco poprawia doświadczenie użytkownika podczas debugowania błędów składniowych.

4.3.5. Strumieniowe przetwarzanie wejścia

W celu unikania nadmiernego zużycia pamięci, lekser analizuje wejście w sposób strumieniowy poprzez implementację interfejsu `CharSequence`. Klasa `LazyReader` realizuje tę funkcjonalność: pobiera dane ze źródła w blokach o rozmiarze 16 KB (ang. *chunks*) i buforuje je lokalnie. Metoda `ensure` zapewnia, że żądana pozycja jest dostępna w buforze, czytając kolejne porcje danych w razie potrzeby.

```
1  @tailrec
2  private def ensure(pos: Int): Unit =
3    if pos >= buffer.length then
4      val charsRead = reader.read(chunk)
5      if charsRead == -1 then
6        throw new IndexOutOfBoundsException(s"Position $pos is out of
     bounds for LazyReader of size $size")
7      else
8        buffer.appendAll(chunk.iterator.take(charsRead))
9        ensure(pos)
```

Listing 4.1. Implementacja metody `ensure` w klasie `LazyReader`.

4.3.6. Wczesna walidacja wzorców

Przed wygenerowaniem automatu skanującego, makro kompilacyjne uruchamia moduł `RegexChecker`, który analizuje wzorce pod względem potencjalnych konfliktów. Mechanizm ten wykrywa dwa klasy problemów wynikających z decyzji implementacyjnej o wykorzystaniu kolejności definicji zamiast reguły najdłuższego dopasowania.

Formalne definicje problemów

Niech $P = \{p_1, p_2, \ldots, p_n\}$ będzie uporządkowaną sekwencją wzorców tokenów, gdzie p_i poprzedza p_j dla $i < j$. Niech $L(p)$ oznacza język regularny akceptowany przez wzorzec p.

Subsumpcja

Wzorzec p_j jest subsumowany przez p_i (gdzie $i < j$), gdy:

$$L(p_j) \subseteq L(p_i)$$

W tej sytuacji żaden ciąg zaakceptowany przez p_j nie zostanie nigdy dopasowany, ponieważ wcześniejszy wzorzec p_i zawsze dopasuje się pierwszy. Jest to analogiczne do „martwego kodu” w programowaniu.

Przykład: jeśli definiuje się `ID = [a-z]+`, a następnie `KEYWORD = if|then`, to $L(\text{KEYWORD}) \subseteq L(\text{ID})$, ponieważ słowa `if` i `then` są również akceptowane przez `[a-z]+`. W rezultacie token `KEYWORD` nigdy nie zostanie rozpoznany.

Pokrycie prefiksów

Wzorzec p_i pokrywa prefiksy wzorca p_j (gdzie $i < j$), gdy istnieje słowo $w \in L(p_j)$ takie, że:

$$\exists u, v \in \Sigma^* : w = uv \wedge u \in L(p_i)$$

Innymi słowy, wcześniejszy wzorzec akceptuje prefiks słowa akceptowanego przez późniejszy wzorzec.

Przykład: jeśli zdefiniowano `LT = <` oraz `LE = <=`, to wzorzec `<` akceptuje prefiks `<` słowa `<=`. Lekser dopasuje znak `<` jako token `LT`, pozostawiając znak `=` jako nierozpoznany, co skutkuje błędem leksykalnym zamiast poprawnego rozpoznania tokenu `LE`.

Implementacja algorytmu walidacji

Algorytm walidacji wykorzystuje bibliotekę `dregex` [55], która implementuje działania na językach regularnych i umożliwia sprawdzanie relacji zawierania między wyrażeniami regularnymi.
Algorytm przebiega w następujących krokach:

1. Każdy wzorzec p jest transformowany do postaci $p \cdot \Sigma^*$ (w notacji wyrażeń regularnych: `p.*`), co umożliwia wykrywanie pokrycia prefiksów. Wzorzec p' pokrywa prefiksy p wtedy i tylko wtedy, gdy $L(p) \subseteq L(p' \cdot \Sigma^*)$.
2. Wszystkie rozszerzone wzorce są kompilowane do wewnętrznej reprezentacji biblioteki `dregex`, która konstruuje strukturę danych umożliwiającą efektywne sprawdzanie relacji podzbioru.
3. Dla każdej pary (i, j) takiej, że $i < j$, algorytm sprawdza, czy:

   $$L(p_j \cdot \Sigma^*) \subseteq L(p_i \cdot \Sigma^*)$$

   co odpowiada zarówno subsumpcji, jak i pokryciu prefiksów.
4. Dla każdej pary naruszającej warunek algorytm generuje komunikat `Pattern p_j is shadowed by p_i`

Integracja z procesem kompilacji

Moduł `RegexChecker` jest wywoływany w fazie ekspansji makra `lexer`, przed generacją kodu leksera. Wykrycie któregokolwiek konfliktu powoduje przerwanie kompilacji z obszernym komunikatem diagnostycznym, dzięki czemu błędy konfiguracji są eliminowane przed czasem wykonania programu. Podejście to jest zgodne z zasadą *fail-fast* w inżynierii oprogramowania oraz filozofią języka Scala polegającą na maksymalnym wykorzystaniu systemu typów i walidacji w czasie kompilacji.

Rozdział 5

Algorytmy analizy składniowej

5.1. Teoretyczne podstawy działania parserów

Analizator składniowy przekształca strumień tokenów w strukturę danych (zazwyczaj drzewo składniowe), rozstrzygając zgodność wejścia z zadeklarowaną gramatyką. Klasyczne podejścia opierają się na automatach ze stosem (PDA, ang. *pushdown automaton*) [56] oraz na algorytmach predykcyjnych [57] lub analizie przesuwająco-redukcyjnej [58].

5.1.1. Gramatyki bezkontekstowe i klasy parserów

Gramatyka bezkontekstowa (CFG) definiowana jest poprzez nieterminale, terminale (tokeny), symbol startowy oraz zbiór produkcji. Wśród klas parserów wyróżnia się dwie główne.

LL(k) (ang. *Left-to-right, Leftmost derivation*) [59]—parsery zstępujące, predykcyjne: konstruują lewostronne wyprowadzenia, wybierając produkcje na podstawie prefiksu wejścia (ang. *lookahead*). Wymuszają ograniczenia na gramatykę: brak lewostronnej rekurencji oraz niekolidujące zbiory `FIRST`/`FOLLOW`.

LR(k) (ang. *Left-to-right, Rightmost derivation*) [60]—parsery wstępujące, przesuwająco-redukcyjne: rekonstruują prawostronne wyprowadzenia wstecz, operując na stosie stanów automatu LR. Obsługują szerszą klasę gramatyk, w tym zawierające lewostronną rekurencję.

5.1.2. Modelowanie automatu z stosem

Parser można sformalizować jako deterministyczny automat ze stosem (PDA): stan określa aktualną pozycję w tabelach parsera, stos przechowuje nieterminale i stany pośrednie, a wejście dostarcza sekwencję tokenów. Przejścia realizują dwie operacje: *shift* (przesunięcie tokenu na stos) oraz *reduce* (zastąpienie prawej strony produkcji nieterminalem i przejście do nowego stanu).

5.1.3. Tabele sterujące

Parsery tabelowe (LL i LR) charakteryzują się stałą złożonością czasową na każdy token [61]. Tabela akcji parsera LR mapuje parę (stan, prefiks wejścia) na akcję: *shift*, *reduce*, *accept*, lub *error*. Tabela `goto` określa przejścia po redukcjach do nowych stanów.

W parserach LL analogiczną rolę spełnia tabela predykcji (nieterminal × prefiks wejścia → produkcja) [59].

5.1.4. Rozstrzyganie konfliktów

Konflikty *shift/reduce* i *reduce/reduce* pojawiają się, gdy tabela parsera byłaby niedeterministyczna (wielokrotny wpis dla tej samej pary stan-prefiks wejścia). Rozwiązania tradycyjne obejmują zmianę gramatyki lub zwiększenie prefiksu wejścia [62]. W parserach LL problemy wynikają typowo z lewostronnej rekurencji i wspólnych prefiksów (wymagająca lewostronnej faktoryzacji). W parserach LR konflikty często biorą się ze zbliżonych prefiksów wielu produkcji (np. `if-then` vs. `if-then-else`) oraz z niejednoznaczności wyrażeń arytmetycznych [59].

5.2. Dobór klasy parsera

System ALPACA generuje parsery klasy LR(1)—deterministyczne analizatory wykorzystujące pełny jednoelementowy zbiór prefiksów wejścia oraz kanoniczne stany LR. Wybór ten łączy wysoką moc wyrazu gramatyk (obsługa lewostronnej rekurencji, złożonych struktur składniowych) z przewidywalnym zachowaniem i stabilną wydajnością parsowania.

Zalety podejścia LR(1) są następujące:

- LR(1) obsługuje istotnie więcej konstrukcji niż LL(k) i eliminuje konieczność wymuszonych przekształceń gramatyki (eliminacji lewej rekurencji, agresywnej lewostronnej faktoryzacji). Specyfikacja pozostaje zbliżona do naturalnej formy języka.
- Pełny zbiór prefiksów wejścia eliminuje konflikty typowe dla SLR i LALR wynikające z nadmiernie szerokich zbiorów `FOLLOW`. W rezultacie specyfikacja wymaga mniej manualnych deklaracji rozwiązywania konfliktów.
- Stany LR(1) jawnie wskazują, jaki token był oczekiwany w danym miejscu. Informacja ta umożliwia generowanie precyzyjnych komunikatów błędów składniowych z kontekstem.
- Każdy krok parsera to pojedynczy dostęp do tabeli akcji (operacja *shift*/*reduce*), gwarantując liniową złożoność czasową względem długości wejścia.
- Każda redukcja LR(1) jednoznacznie odpowiada konkretnej produkcji i ma dostęp do wszystkich terminali ją tworzących. Akcje semantyczne mogą zatem bezpośrednio konstruować węzły drzewa składni abstrakcyjnej.

Koszty i ograniczenia obejmują następujące aspekty:

- Kanoniczne LR(1) może generować znacznie więcej stanów niż uproszczone warianty (SLR, LALR), co wpływa na rozmiar tabeli akcji.
- Obliczanie zbiorów domknięcia (ang. *closure*) i przejść (ang. *goto*) dla LR(1) wymaga więcej obliczeń niż w uproszczonych wariantach, wpływając na czas kompilacji oraz rozmiar generatora [63].
- Parser jest deterministyczny na poziomie składniowym, lecz wciąż muszą zostać rozwiązane niejednoznaczności (np. *dangling else*) poprzez jawne reguły precedencji operatorów.

- Choć klasa jest szeroka, nadal istnieją konstrukcje wymagające przekształceń w celu dopasowania do ograniczeń LR(1).

5.3. Konstrukcja tabel parsera LR(1)

Generowanie parsera LR(1) w ALPACA stanowi sekwencję algorytmów realizowanych w czasie kompilacji. W sekcji 5.3.1 opisano, etap po etapie, transformację deklaratywnej specyfikacji gramatyki w deterministyczną tabelę akcji.

5.3.1. Wyznaczanie zbiorów FIRST

Na podstawie produkcji obliczany jest zbiór `FirstSet` (iteracyjnie do osiągnięcia punktu stałego) [64]. `FirstSet`[N] zawiera wszystkie możliwe terminale, które mogą pojawić się na początku wyprowadzenia z nieterminala N.
Algorytm iteruje po wszystkich produkcjach, dodając:

- pierwszy terminal z prawej strony produkcji,
- w przypadku nieterminala—wszystkie terminale ze zbioru `FIRST` tego nieterminala (z wyjątkiem symbolu pustego ε); jeśli nieterminal może generować ε, algorytm kontynuuje do kolejnych symboli produkcji,
- symbol ε dla produkcji mogących generować pusty ciąg.

Pętla powtarza się do osiągnięcia punktu stałego (kiedy `FIRST` przestają się zmieniać).

```
@tailrec
private def addImports(firstSet: FirstSet, production: Production):
 FirstSet = production match {
  case Production.NonEmpty(lhs, NEL(head: Terminal, tail)) =>
    firstSet.updated(lhs, firstSet(lhs) + head)

  case Production.NonEmpty(lhs, NEL(head: NonTerminal, tail)) =>
    val newFirstSet = firstSet.updated(lhs, firstSet(lhs) ++
(firstSet(head) - Symbol.Empty))

    val nextProduction = tail match
      case head :: next => Production.NonEmpty(lhs, NEL(head, next*))
      case Nil => Production.Empty(lhs)

    if firstSet(head).contains(Symbol.Empty)
    then addImports(newFirstSet, nextProduction)
    else newFirstSet

  case Production.Empty(lhs) =>
    firstSet.updated(lhs, firstSet(lhs) + Symbol.Empty)
}
```

Listing 5.1. Implementacja metody obliczającej zbiory FIRST.

5.3.2. Budowa automatów LR(1)

Stan początkowy to domknięcie elementu `S' → • root, $`, gdzie S' jest generowanym symbolem startowym, root jest symbolem startowym gramatyki, a $ reprezentuje koniec

wejścia. Kolejne stany konstruowane są klasycznym schematem *closure/goto* [65] i deduplikowane, aby otrzymać deterministyczny automat LR(1).

Funkcja closure

Dla elementu zawierającego nieterminal po kropce $(A \to \alpha \bullet B\,\beta\ ,\ a)$, funkcja `closure` realizuje następujące kroki:

1. Oblicza zbiór możliwych prefiksów wejścia: $FIRST(\beta\,a) = FIRST(\beta)$ bez ε, a jeśli $\varepsilon \in FIRST(\beta)$, dodaje także a.
2. Dla każdego prefiksu wejścia x dodaje elementy $B \to \bullet\,\gamma\ ,\ x$ dla wszystkich produkcji $B \to \gamma$.
3. Rekurencyjnie domyka nowo dodane elementy, kontynuując proces, dopóki nie zostaną dodane nowe produkcje (punkt stały domknięcia).

W rezultacie stan zawiera pełny zbiór przewidywań dla wszystkich nieterminali, które mogą pojawić się w tej pozycji.

```
def closure(
    state: State,
    item: Item,
    productions: List[Production],
    firstSet: FirstSet
): State =
  if !item.isLastItem && !item.nextSymbol.isInstanceOf[Terminal] then
    val lookAheads = item.nextTerminals(firstSet)

    productions.view
      .filter(_.lhs == item.nextSymbol)
      .foldLeft(state + item) { (acc, production) =>
        lookAheads.foldLeft(acc) { (acc, lookAhead) =>
          val item = production.toItem(lookAhead)

          if state.contains(item) then acc
          else closure(acc, item, productions, firstSet)
        }
      }
  else state + item
```

Listing 5.2. Implementacja funkcji `closure`.

Funkcja goto

Dla zadanego stanu i symbolu s po kropce funkcja `goto` realizuje przesunięcie kropki we wszystkich elementach zawierających ten symbol, a następnie stosuje `closure` do wyniku, korzystając z wcześniej obliczonych zbiorów `FIRST`.

```
def goto(
    state: State,
    step: Symbol,
    productions: List[Production],
    firstSet: FirstSet
): State =
  state.view
    .filter(item => !item.isLastItem && item.nextSymbol == step)
    .foldLeft(State.empty) { (acc, item) =>
      closure(acc, item.nextItem, productions, firstSet)
    }
```

Listing 5.3. Implementacja funkcji `goto`.

Główny algorytm budowy LR(1)

Konstrukcja automatu LR(1) rozpoczyna się od stanu początkowego i iteracyjnie dodaje nowe stany, dopóki wszystkie możliwe przejścia nie zostaną odkryte. Proces można podzielić na cztery główne etapy:

1. Domknięcie elementu `S' → • root, $` tworzy stan 0 (stan początkowy automatu).
2. Dla każdego stanu zbierane są symbole, które mogą pojawić się po kropce. Dla każdego takiego symbolu obliczany jest stan docelowy za pomocą funkcji `goto`. Jeśli stan docelowy już istnieje, do tabeli akcji dodawana jest akcja *shift* do jego ID; w przeciwnym razie stan otrzymuje nowe ID, zostaje dodany do listy stanów, a *shift* wskazuje na nowy stan.
3. Elementy z kropką na końcu produkcji (produkcja całkowicie rozpoznana) dodają akcje *reduce* dla swoich prefiksów wejścia. Specjalny element `S' → root •, $` generuje akcję *accept* dla symbolu `$`.
4. Algorytm powtarza powyższe kroki dla wszystkich wygenerowanych stanów. Identyczne zestawy elementów nie tworzą nowych stanów, gwarantując, że automat pozostaje deterministyczny i skończony.

```
var currStateId = 0

val initialState =
  closure(
    State.empty,
    productions.find(_.lhs == parser.Symbol.Start).get.toItem(),
    // S' -> • root, $
    productions,
    firstSet
  )

val states = mutable.ListBuffer(initialState)
val table = mutable.Map.empty[(state: Int, stepSymbol: Symbol),
   ParseAction]

while states.sizeIs > currStateId do
  val currState = states(currStateId)

```

```
18  // reductions and acceptations
19  for item <- currState if item.isLastItem do
20      addToTable(item.lookAhead, Reduction(item.production))
21
22  // shift (goto) transitions
23  for stepSymbol <- currState.possibleSteps do
24    val newState = goto(currState, stepSymbol, productions, firstSet)
25
26    states.indexOf(newState) match
27      case -1 =>
28        val newId = states.length
29        addToTable(stepSymbol, Shift(newId))
30        states += newState
31      case stateId =>
32        addToTable(stepSymbol, Shift(stateId))
33
34  currStateId += 1
```

Listing 5.4. Główny algorytm budowy automatów LR(1).

Stany są przechowywane w posortowanych zbiorach strukturalnych, dzięki czemu są porównywane na podstawie zawartości (nie referencji), co umożliwia naturalną deduplikację.

5.3.3. Rozwiązywanie konfliktów parsera

Podczas konstrukcji tabel parsera LR(1) mogą wystąpić sytuacje, w których dla tej samej pary (stan, symbol) istnieje więcej niż jedna możliwa akcja—konflikty *shift/reduce* lub *reduce/reduce*. W takich przypadkach parser musi posiadać mechanizm rozstrzygania, która akcja ma pierwszeństwo.

Podstawowy mechanizm precedencji

ALPACA implementuje system deklaratywnej precedencji oparty na relacjach pierwszeństwa między produkcjami i tokenami. Użytkownik definiuje zbiór reguł precedencji, z których każda wskazuje, że dana produkcja lub token ma pierwszeństwo nad inną produkcją lub tokenem. Te reguły są przechowywane w tabeli rozwiązywania konfliktów (ang. *conflict resolution table*)—strukturze mapującej każdy klucz (produkcja lub nazwa tokenu) na zbiór kluczy, nad którymi ma on pierwszeństwo.

Algorytm rozstrzygania konfliktu

Podczas budowy tabeli parsera, gdy algorytm napotyka konflikt między akcjami a_1 i a_2, realizowane są następujące kroki.

1. Sprawdź, czy a_1 poprzedza a_2.
2. Jeśli nie, sprawdź czy a_2 poprzedza a_1.
3. Jeśli brak relacji precedencji, zgłoś błąd kompilacji z komunikatem diagnostycznym.

W przypadku braku reguły precedencji kompilator rzuca wyjątek zawierający następujące informacje.

- opis konfliktu (shift/reduce lub reduce/reduce),

- symbol wywołujący konflikt,
- ścieżkę w gramatyce prowadzącą do konfliktu,
- sugesitę użycia mechanizmu `before`/`after`.

Przechodniość przez rozwinięcie grafu relacji

Kluczowym udoskonaleniem systemu jest automatyczne zapewnienie przechodniości relacji precedencji. Zamiast wymagać od użytkownika jawnego zdefiniowania wszystkich par relacji (co byłoby uciążliwe przy większych gramatykach), ALPACA traktuje reguły precedencji jako graf skierowany. Relacja między dwoma kluczami jest uznawana za istniejącą, jeśli istnieje między nimi ścieżka w grafie. Gdy parser próbuje rozstrzygnąć konflikt między akcjami a_1 i a_2, system wykonuje przeszukiwanie grafu w głąb (ang. *depth-first search*, DFS). Jeśli w trakcie przeszukiwania odnajdzie ścieżkę między akcjami, oznacza to, że istnieje (bezpośrednia lub przechodnia) precedencja między nimi. Algorytm śledzi odwiedzone węzły, aby uniknąć zapętlenia podczas przeszukiwania.

Dzięki temu rozwiązaniu użytkownik może zadeklarować jedynie podstawowe reguły precedencji (np. *mnożenie* $\prec$ *dodawanie*, gdzie $\prec$ oznacza wyższy priorytet), a system automatycznie wyprowadzi wszystkie implikacje przechodnie. Podejście to drastycznie redukuje liczbę deklaracji wymaganych od użytkownika, zachowując przy tym pełną kontrolę nad rozwiązywaniem konfliktów.

Przykład: Dla gramatyki wyrażeń arytmetycznych z reguł:

$$\texttt{pow} \prec \texttt{mul}$$
$$\texttt{mul} \prec \texttt{add}$$

system automatycznie wyprowadza $\texttt{pow} \prec \texttt{add}$ poprzez ścieżkę $\texttt{pow} \to \texttt{mul} \to \texttt{add}$.

Dzięki temu użytkownik definiuje jedynie $n-1$ reguł dla n poziomów precedencji (zamiast $\binom{n}{2}$ reguł bez przechodniości).

```
def getHigherPrecedenceAction(first: ParseAction, second: ParseAction):
    Option[ParseAction] = {
  def winsOver(first: ParseAction, second: ParseAction):
    Option[ParseAction] = {
     @tailrec
     def loop(queue: List[ParseAction], visited: Set[ParseAction]):
    Option[ParseAction] = queue match
       case Nil => None
       case `second` :: _ => Some(first)
       case head :: tail =>
         val current = table.getOrElse(head, Set.empty)
         val neighbors = current.diff(visited)
         loop(tail ++ neighbors, visited + head)

    loop(List(first), Set())
  }

  winsOver(first, second) orElse winsOver(second, first)
}
```

Listing 5.5. Implementacja rozwiązywania konfliktów z przechodniością.

Weryfikacja acykliczności w czasie kompilacji

Istotnym wymogiem poprawności systemu precedencji jest brak cykli w grafie relacji. Obecność cyklu oznaczałaby niespójność, w której produkcja A miałaby pierwszeństwo nad B, B nad C, a jednocześnie C nad A—sytuację nie do rozstrzygnięcia.

ALPACA weryfikuje acykliczność grafu precedencji w czasie kompilacji za pomocą algorytmu DFS z kolorowaniem węzłów [66]. Każdy węzeł może znajdować się w jednym z trzech stanów:

- `Unvisited` — węzeł jeszcze nieodwiedzony
- `Visited` — węzeł w trakcie eksploracji, znajduje się na aktualnej ścieżce DFS
- `Processed` — węzeł całkowicie przetworzony, wszystkie jego następniki zostały sprawdzone

```
def verifyNoConflicts(): Unit = {
  enum VisitState:
    case Unvisited, Visited, Processed

  enum Action:
    case Enter(node: ConflictKey, path: List[ConflictKey] = Nil)
    case Leave(node: ConflictKey)

  val visited = mutable.Map.empty[ConflictKey,
   VisitState].withDefaultValue(VisitState.Unvisited)

  @tailrec
  def loop(stack: List[Action]): Unit = stack match
    case Nil => // Done

    case Action.Leave(node) :: rest =>
      visited(node) = VisitState.Processed
      loop(rest)

    case Action.Enter(node, path) :: rest =>
      visited(node) match
        case VisitState.Processed => loop(rest)
        case VisitState.Visited => throw InconsistentConflictResolution()
        case VisitState.Unvisited =>
          visited(node) = VisitState.Visited
          val neighbors = table.getOrElse(node,
   Set.empty).map(Action.Enter(_, node :: path))
          loop(neighbors ::: List(Action.Leave(node)) ::: rest)

  for node <- table.keys do loop(Action.Enter(node) :: Nil)
}
```

Listing 5.6. Implementacja weryfikacji acykliczności grafu precedencji.

Kluczowe właściwości

- Jeśli algorytm zakończy się bez komunikatu o błędzie, graf jest acykliczny (DAG)
- Każdy cykl zostanie wykryty
- Każdy węzeł i krawędź odwiedzane dokładnie raz — Złożoność czasowa $O(|V|+|E|)$

Jeśli algorytm napotka węzeł w stanie `Visited` podczas eksploracji, oznacza to wykrycie cyklu. Kompilator informuje o błędzie `InconsistentConflictResolution` zawierającym następujące elementy:

- Węzeł gdzie wykryto cykl
- Ścieżkę prowadzącą do cyklu
- Pełny cykl
- Sugestię rewizji reguł `before`/`after`

```
Inconsistent conflict resolution detected:
Reduction(sub) before Shift(*) before Reduction(sub)
There are elements being both before and after Reduction(sub)
at the same time.
Consider revising the before/after rules to eliminate cycles
```

Listing 5.7. Przykładowy komunikat o konfliktach.

Weryfikacja ta zapewnia, że wszelkie błędy w deklaracjach precedencji są wykrywane na etapie kompilacji gramatyki, eliminując możliwość nieokreślonego zachowania parsera w czasie wykonania. Jest to kluczowy element zasady *fail-fast*, która maksymalizuje bezpieczeństwo i przewidywalność systemu.

Rozdział 6

Organizacja pracy

Rozdział przedstawia metodykę realizacji projektu dyplomowego *ALPACA*, jego charakter, podział obowiązków pomiędzy członkami zespołu, organizację prac oraz zastosowane techniki inżynierskie. Szczególny nacisk położono na charakterystykę procesu wytwórczego, narzędzia wspierające współpracę zespołową oraz strategię walidacji implementacji.

6.1. Charakterystyka projektu i sposób realizacji

Projekt *ALPACA* ma charakter badawczo-rozwojowy i łączy elementy badań teoretycznych z praktyczną implementacją. Podstawowe wymaganie projektu, polegające na implementacji narzędzia umożliwiającego generowanie analizatorów leksykalnych i składniowych w fazie kompilacji przy pełnym wsparciu środowiska IDE, zostało jedynie częściowo sprecyzowane na etapie początkowym. W dalszych etapach realizacji specyfikacja była doprecyzowywana w sposób iteracyjny, na podstawie wyników eksperymentów oraz analizy ograniczeń technicznych.

W pierwszym semestrze zrealizowano fazę eksploracyjno-prototypową, obejmującą analizę możliwości metaprogramowania w Scali 3, prototypowanie mechanizmów generacji leksera i parsera oraz identyfikację ograniczeń technicznych platformy JVM. W drugim semestrze zrealizowano fazę wdrożeniowo-optymalizacyjną, skoncentrowaną na utrwaleniu wybranych rozwiązań, rozszerzeniu funkcjonalności systemu oraz poprawie jakości i wydajności kodu.

6.2. Zespół i podział obowiązków

6.2.1. Osoby w projekcie i ich role

Projekt realizowany był przez zespół dwóch osób o jasno zdefiniowanych rolach.

Bartosz Buczek (dalej: BB) był odpowiedzialny przede wszystkim za implementację algorytmów analizy leksykalnej i składniowej, opracowanie mechanizmu rozwiązywania konfliktów gramatycznych oraz przygotowanie testów wydajnościowych.

Bartłomiej Kozak (dalej: BK) odpowiadał za implementację języka dziedzinowego (DSL) systemu *ALPACA* z wykorzystaniem metaprogramowania w Scali 3, generację tabel parsowania w fazie kompilacji, projektowanie zaawansowanych mechanizmów systemu typów oraz opracowanie dokumentacji.

6.2.2. Podział prac na główne zadania

Projekt podzielono na następujące obszary funkcjonalne, z przypisaniem głównych odpowiedzialności.

System leksykalny

Zakres:

- Implementacja makra `lexer` transformującego deklaratywne reguły tokenizacji w kod proceduralny. (BK)
- Projektowanie interfejsu DSL opartego na funkcjach częściowych. (BB, BK)
- Integracja z wyrażeniami regularnymi biblioteki standardowej Scali. (BB)
- Definiowanie typów rafinowanych dla tokenów. (BK)
- Obsługa ignorowanych reguł leksykalnych. (BB)
- Obsługa kontekstu w lekserze. (BB, BK)
- Diagnostyka błędów leksykalnych. (BB, BK)

Artefakty:

- Moduł `alpaca.lexer`
- Klasy: `LexerDefinition`, `Tokenization[Ctx]`, `DefinedToken`
- Makra: `lexer` oraz narzędzia pomocnicze (`CompileNameAndPattern`, `ReplaceRefs`)

System parserów

Zakres:

- Implementacja algorytmów konstrukcji stanów LR(1) w fazie kompilacji. (BB, BK)
- Generacja tabel parsowania (tabela akcji, tabela `goto`). (BB)
- Transformacja akcji semantycznych z kontekstu definicji do wygenerowanych tabel. (BK)
- Obsługa ograniczeń JVM (fragmentacja metod, limit rozmiaru). (BK)
- Deklaratywne rozwiązywanie konfliktów *shift–reduce* i *reduce–reduce*. (BB, BK)
- Obsługa kontekstu w parserze. (BB, BK)
- Diagnostyka błędów składniowych. (BB, BK)

Artefakty:

- Moduł `alpaca.parser`
- Klasy: `Parser[Ctx]`, `Rule[R]`, `ParseTable`, `ActionTable`
- Makra: `createTables`

Infrastruktura i narzędzia pomocnicze

Zakres:

- Implementacja pomocniczych klas i makr, m.in. `Empty[T]`, `ReplaceRefs`, `CreateLambda`, `Copyable[T]`. (BK)

- Przygotowanie systemu testów jednostkowych i integracyjnych. (BB)
- Konfiguracja procesu budowania projektu (system `mill`). (BK)
- Dokumentacja techniczna. (BB, BK)

Dokumentacja pracy dyplomowej

Zakres:

- Cel pracy i wizja projektu. (BB, BK)
- Metaprogramowanie w Scali 3. (BK)
- Implementacja. (BK)
- Algorytmy analizy leksykalnej. (BB)
- Algorytmy analizy składniowej. (BB)
- Analiza porównawcza z istniejącymi rozwiązaniami. (BB)
- Organizacja pracy. (BK)

6.2.3. Współpraca między członkami zespołu

Pomimo wyraźnego podziału obowiązków, współpraca między członkami zespołu miała charakter ścisły i ciągły. Każdy pull request do repozytorium był poddawany przeglądowi kodu przez drugiego członka zespołu przed włączeniem zmian do głównej gałęzi. Zadania rejestrowano w postaci zgłoszeń *GitHub Issues* i organizowano na tablicy Kanban[67], a kamienie milowe wraz z terminami realizacji definiowano i monitorowano z wykorzystaniem mechanizmu GitHub Milestones[68]. Wspólnie projektowano interfejsy między modułami (między innymi interfejs `Token` oraz parametryzację `Ctx`) oraz rozwiązywano problemy techniczne wymagające wiedzy o różnych komponentach systemu. Prace prowadzono z wykorzystaniem systemu kontroli wersji Git.

6.3. Organizacja prac i wykorzystane narzędzia

6.3.1. Komunikacja zespołowa

Regularne spotkania

- Spotkania zespołu odbywały się w interwałach 2–3-dniowych, głównie w formie asynchronicznej z wykorzystaniem komunikatorów.
- Sesje debugowania organizowano w miarę potrzeby, w sytuacjach wymagających jednoczesnej pracy obu członków zespołu.
- Burze mózgów, m.in. dotyczące projektowania DSL, realizowano w formie spotkań ad hoc.

Kanały komunikacji

Komunikacja zespołu opierała się na kilku komplementarnych kanałach. Głównym narzędziem do dyskusji nad kodem, proponowania zmian oraz rejestrowania błędów były

GitHub Issues i Pull Requests. Signal służył jako kanał komunikacji tekstowej dla szybkich pytań oraz wymiany odnośników i materiałów. Spotkania osobiste wykorzystywano przede wszystkim do omówień strategicznych i podejmowania decyzji architektonicznych.

Ustalanie kamieni milowych

Podział prac na etapy został sformalizowany poprzez zdefiniowanie pięciu kamieni milowych (ang. *milestones*), każdego powiązanego z wyraźnym terminem i katalogiem zadań. Kamienie milowe pozwoliły zespołowi na śledzenie postępów, priorytetyzowanie prac oraz szybkie identyfikowanie zagrożeń dla harmonogramu.

Każdy kamień milowy zawierał listę zgłoszeń (ang. *issues*) reprezentujących konkretne zadania (implementacja funkcji, dokumentacja, testy). Postęp mierzono poprzez stosunek zamkniętych zgłoszeń do całkowitej liczby planowanych zadań, co umożliwiało szybką ocenę stanu rzeczywistego realizacji w stosunku do pierwotnego planu.

Strukturę kamieni milowych oraz postępy realizacji przedstawiono w tabel 6.1.

Kamień milowy	**Opis**	**Termin**	**Postęp**
MVP	Implementacja podstawowych funkcjonalności leksera i parsera, minimalne działające rozwiązanie.	30 września 2025	15/15
Core Features	Rozszerzenie funkcjonalności.	31 października 2025	22/25
Stretch Goals	Funkcjonalności dodatkowe (optymalizacje, rozszerzona diagnostyka błędów, dodatkowe przykłady)	brak ustalonego terminu	21/30
Testing & Benchmarking	Kompleksowe testowanie (testy integracyjne, benchmarki wydajności, walidacja gramatyk)	30 listopada 2025	2/5
Thesis	Finalizacja rozprawy dyplomowej (pisanie rozdziałów, bibliografia, ostateczna redakcja)	15 grudnia 2025	12/13

Tabela 6.1. Kamienie milowe projektu *ALPACA* i postęp ich realizacji.

Postęp realizacji wskazuje, że zespół pomyślnie ukończył pierwsze dwa kamienie milowe w wyznaczonym terminie, trzecia faza (Stretch Goals) była realizowana równolegle i bieżąco, czwarta faza wymagała intensywniejszych wysiłków na ostatnim etapie projektu, natomiast piąta faza (Thesis) była finalizowana w ostatnich tygodniach przed terminem oddania pracy dyplomowej.

6.3.2. Narzędzia programistyczne i CI/CD

System kontroli wersji Git i GitHub

Całość projektu hostowana jest na platformie GitHub[69].

Narzędzie do budowania — `mill` 1.x

Konfiguracja w pliku `build.mill` definiuje zależności, wersję kompilatora Scali (3.8.0) oraz dodatkowe pluginy. Typowe polecenia obejmują `mill compile`, `mill test` oraz `mill run`. Średni czas kompilacji projektu wynosi około 45 sekund (w tym około 35 sekund na uruchomienie testów).

Weryfikacja jakości

Weryfikacja jakości kodu odbywała się z wykorzystaniem narzędzia Scalafmt, zapewniającego automatyczne formatowanie kodu do spójnego stylu. Konfigurację utrzymywano w pliku `.scalafmt.conf`.

Dodatkowo projekt kompiluje się bez ostrzeżeń przy włączonych zaostrzonych opcjach kompilatora, takich jak `-Xfatal-warnings` (traktowanie ostrzeżeń jako błędów) czy `-Ycheck:macros` (dodatkowa weryfikacja poprawności makr).

Testowanie

Wykorzystano framework testowy *ScalaTest*, który umożliwił przygotowanie testów jednostkowych dla modułów leksera i parsera, a także testów integracyjnych dla pełnych przykładów (parser wyrażeń arytmetycznych, parser uproszczonego formatu JSON). Testy uruchamiano poleceniem `mill test`. Pokrycie testami obejmuje główne ścieżki wykonania i przypadki brzegowe.

Testowanie systemów opartych na makrach kompilacyjnych stanowi szczególne wyzwanie, ponieważ makra wykonywane są w fazie kompilacji, a błędy mogą ujawniać się dopiero na poziomie wygenerowanego kodu. W projekcie zastosowano zarówno testy pozytywne, weryfikujące poprawną kompilację oraz semantykę wygenerowanego kodu, jak i testy negatywne, sprawdzające poprawne odrzucanie niepoprawnych definicji z odpowiednimi komunikatami diagnostycznymi. Dodatkowo przygotowano testy wydajnościowe oraz scenariusze integracyjne typu end-to-end (obejmujące jednocześnie lekser i parser), aby ocenić zachowanie systemu w warunkach zbliżonych do rzeczywistego użycia.

Continuous Integration z wykorzystaniem GitHub Actions

Każde wypchnięcie zmian do repozytorium uruchamia przepływ pracy CI, obejmujący kompilację projektu, uruchomienie testów, weryfikację formatowania kodu (Scalafmt) oraz prezentację statusu tych kroków na pull requestach przed ich połączeniem z głównym branchem.

6.3.3. Dokumentacja

Dokumentacja w kodzie

Każda publiczna klasa, funkcja oraz makro opatrzone są komentarzami Scaladoc. Komentarze dokumentują cel, parametry, wartości zwracane oraz przykładowe scenariusze użycia. Złożone implementacje (np. algorytm LR(1)) uzupełniono o komentarze liniowe objaśniające kluczowe fragmenty logiki.

Plik `README.md` zawiera instrukcje instalacji, przykłady użycia oraz zarys architektury systemu.

`GitHub Pages` hostuje bardziej szczegółową dokumentację, poradniki oraz sekcję FAQ[70].

Praca dyplomowa (LaTeX)

Praca została przygotowana w systemie składu LaTeX z wykorzystaniem szablonu AGH (`aghengthesis`). Treść została podzielona na rozdziały w oddzielnych plikach (m.in. `introduction.tex`, `metaprogramming.tex`, `implementation.tex`). Bibliografię przygotowano w formacie BibLaTeX (`bibliografia.bib`) z odwołaniami do literatury naukowej oraz dokumentacji technicznej.

6.4. Zastosowane techniki i praktyki inżynierskie

6.4.1. Metodologia wytwarzania oprogramowania

Iteracyjne podejście do projektowania

Projekt nie posiadał sztywnej, kompletnej specyfikacji funkcjonalnej na etapie początkowym. Projektowanie przebiegało iteracyjnie, zgodnie z następującym schematem:

1. Prototypowanie — szybkie eksperymentowanie z różnymi podejściami.
2. Ewaluacja — ocena wydajności, przydatności oraz spójności otrzymanych rozwiązań.
3. Udoskonalanie — ulepszanie wybranych podejść.
4. Integracja — łączenie poszczególnych komponentów w spójny system.

Test-Driven Development (TDD) — częściowe zastosowanie

Dla modułów o jasno zdefiniowanych specyfikacjach (np. funkcji `computeFirstSet`) testy przygotowywano przed implementacją, zgodnie z podejściem Test-Driven Development. Dla komponentów o charakterze eksploracyjnym (np. mechanizmów generacji klas w makrach) testy opracowywano po ustabilizowaniu implementacji. Praktyka ta była szczególnie przydatna do ujawniania problemów w interfejsach pomiędzy modułami.

Refaktoryzacja

Regularnie przeprowadzano refaktoryzacje w celu poprawy czytelności, jakości oraz efektywności kodu. Uzasadnione zmiany refaktoryzacyjne dokumentowano w dedykowanej sekcji Pull Requesta. Wykorzystywano narzędzia wspomagające refaktoryzację (m.in. Scalafix).

Code Review i Pair Programming

Każdy Pull Request był recenzowany przez drugiego członka zespołu przed połączeniem z głównym branchem. W przypadku zagadnień o szczególnie złożonym charakterze zastosowano sesje wspólnego programowania (ang. *Pair Programming*). Spotkania poświęcone przeglądom kodu odbywały się 2–3 razy w tygodniu, zwykle przez 30–60 minut.

Zgłoszenia i planowanie

Podczas cyklicznych spotkań omawiano priorytety oraz wybierano kolejne zgłoszenia (ang. *issues*) do realizacji. Backlog utrzymywano w postaci zgłoszeń *GitHub Issues*.

6.4.2. Praktyki specyficzne dla metaprogramowania

Obliczenia wieloetapowe (ang. *Staged Computation*)

Świadomie rozróżniano obliczenia wykonywane na etapie kompilacji i na etapie wykonania. Maksymalizowano zakres obliczeń przeniesionych do etapu kompilacji (m.in. generacja tabel LR(1), kompilacja wyrażeń regularnych). Minimalizowano nakład obliczeń w fazie wykonania, co pozwoliło uzyskać wysoką wydajność uruchomieniową.

Walidacja w fazie kompilacji (Validation at Compile-Time)

Walidowano gramatyki już na etapie kompilacji (m.in. wykrywanie konfliktów LR, niepoprawnej składni). Walidowano wyrażenia regularne w definicjach leksera. Stosowano zasadę szybkiego zwrotu błędu (ang. *fail-fast*) — preferowano odrzucanie niepoprawnych danych na etapie kompilacji zamiast zgłaszania błędów dopiero w czasie wykonania.

6.4.3. Praktyki DevOps

Continuous Integration

GitHub Actions automatyzowało kompilację, uruchamianie testów oraz weryfikację formatowania kodu przy każdym wypchnięciu zmian. Zastosowano reguły ochrony gałęzi (ang. *branch protection rules*), wymagające pozytywnego wyniku wszystkich testów przed połączeniem zmian z głównym branchem.

Zarządzanie artefaktami (Artifact Management)

Wersje rozwojowe (SNAPSHOT) i stabilne wydania są publikowane w repozytorium Maven Central.

6.5. Przebieg prac — harmonogram i iteracje

6.5.1. Szczegółowa oś czasu i przebieg prac

Praca ALPACA realizowana była przez 27 tygodni, obejmujących dwa semestry akademickie. Szczegółowy przebieg oraz liczba zmian w poszczególnych etapach dokumentowane były poprzez system kontroli wersji, umożliwiający śledzenie postępów na poziomie pojedynczych zmian.

W toku realizacji projektu zebrano 216 commitów (zmian w kodzie), z których średnio przypadało 19,6 zmian na tydzień. Tabela 6.2 prezentuje podział prac na etapy wraz z kluczowymi osiągnięciami i liczbą zmian wprowadzonych w każdym okresie.

Etap	Okres czasowy	Kluczowa aktywność
Inicjalizacja	19 kwietnia 2025	Konfiguracja repozytorium, system budowania Mill
Semestr 1 — Eksploracja i prototypowanie (tygodnie 1–14)		
Tydzień 1–3	22–27 lipca	Podstawy teoretyczne, pierwsze makra, API metaprogramowania
Tydzień 4–6	27 lipca–6 sierpnia	Eksperymentowanie: DFA vs. wyrażenia regularne, framework testowy
Tydzień 7–10	6–25 sierpnia	Szkic algorytmu LR(1), infrastruktura testowa, integracja
Tydzień 11–14	25 września–6 października	Identyfikacja ograniczeń JVM, planowanie semestr 2
Semestr 2 — Implementacja i optymalizacja (tygodnie 15–27)		
Tydzień 15–17	13 października–30 listopada	Moduł leksera, typy rafinowane, interfejs DSL
Tydzień 18–20	30 października–20 listopada	Generator parserów LR(1), obsługa ograniczeń JVM
Tydzień 21–23	20–27 listopada	Integracja leksera i parsera, akcje semantyczne
Tydzień 24–25	27 listopada–5 grudnia	Optymalizacja wydajności, poprawa diagnostyki błędów
Tydzień 26–27	6–15 grudnia	Finalizacja rozprawy dyplomowej, ostateczne poprawki

Tabela 6.2. Szczegółowa oś czasu projektu ALPACA z liczbą zmian (commitów) w poszczególnych etapach.

Charakterystyka poszczególnych etapów

Faza eksploracyjna (tygodnie 1–14) Pierwsza połowa prac skupiła się na zdobyciu doświadczenia z metaprogramowaniem w Scali 3 oraz eksperymentowaniem z różnymi podejściami do generacji leksera. W tym okresie zespół zebrał 55 commitów, co wskazuje na intensywne prototypowanie i ciągłe eksperymentowanie. Szczególnie intensywny okazał się okres tygodni 11–14 (20 commitów), kiedy identyfikowano ograniczenia platformy JVM i opracowywano strategie ich obejścia.

Faza wdrożeniowa (tygodnie 15–25) Druga część projektu cechowała się szybszym tempem zmian (130 commitów), co odzwierciedlało przejście z fazy eksploracji na implementację. Szczególnie dynamiczny był okres tygodni 15–17 (45 commitów), kiedy równocześnie realizowano moduł leksera oraz wprowadzano obsługę typów rafinowanych. Okres tygodni 21–23 (35 commitów) skupił się na integracji komponentów i obsłudze akcji semantycznych w parserze.

Finalizacja (tygodnie 26–27) Ostatnie dwa tygodnie projektu zawierały 15 commitów poświęconych przede wszystkim dokumentacji, ostatecznym poprawkom rozprawy dyplomowej oraz czyszczeniu kodu.

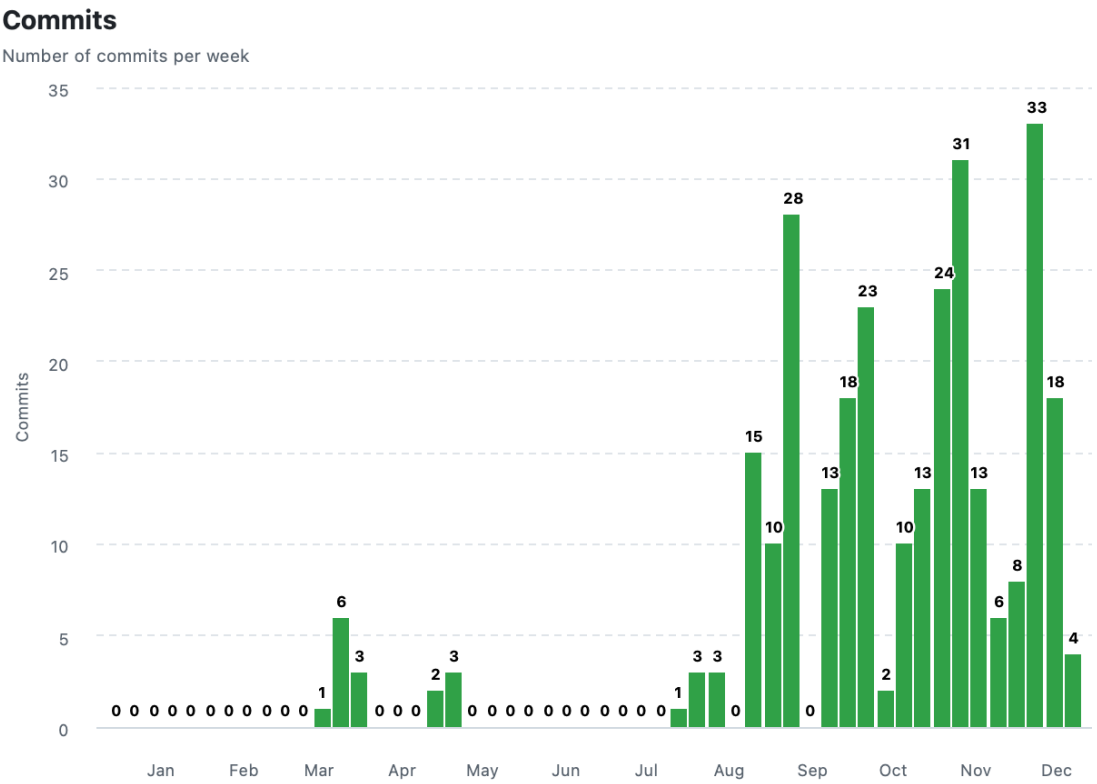
6.5.2. Szybkość pracy i postępy

Metryki wydajności

Realizacja projektu ALPACA wymagała znaczącego zaangażowania zespołu przez okres kilku miesięcy. Skala pracy oraz złożoność napotkanych wyzwań technicznych znajdują odzwierciedlenie w poniższych wskaźnikach:

- Historia rozwoju reprezentuje łącznie 450 commitów zintegrowanych w głównej gałęzi projektu, dokumentujących iteracyjny proces implementacji oraz poprawiania systemu.
- Struktura zmian kodowych obejmuje około 80 Pull Requestów, z których każdy zawierał recenzję i testy weryfikujące poprawność dodawanych funkcjonalności.
- Rozmiar implementacji wynosi około 4000 linii kodu implementacyjnego (bez uwzględnienia testów) oraz około 2000 linii kodów testowych.

Rozkład pracy w czasie pokazany na rysunku 6.1 ilustruje dynamikę zespołu, wskazując na stały postęp oraz okresy intensywnej pracy nad rozwiązywaniem kluczowych wyzwań technicznych.



Rysunek 6.1. Rozkład commitów w tygodniach realizacji projektu ALPACA.

6.6. Główne problemy i ich rozwiązania

6.6.1. Limit rozmiaru metody JVM

Problem

Przy naiwnej implementacji generowanie dużych tabel parsowania LR(1) prowadziło do przekroczenia limitu 64 KB na rozmiar kodu bajtowego pojedynczej metody w JVM.

Przyczyna

Każdy wpis tabeli parsowania (para: stan, symbol terminalny → akcja *shift*/*reduce*) wymagał wygenerowania kilku instrukcji. Dla złożonych gramatyk z tysiącami wpisów rozmiar wygenerowanej metody przekraczał dopuszczalny limit.

Rozwiązanie

Zastosowano podejście polegające na fragmentacji metod (*method fragmentation*) [42]. Zamiast generować pojedynczą metodę zawierającą całą tabelę parsowania, generowanych jest wiele metod pomocniczych, z których każda odpowiada za dodanie ograniczonej liczby wpisów do struktury budującej tabelę. Metoda główna sekwencyjnie wywołuje metody pomocnicze, a następnie zwraca wynikową tabelę. Ze względu na niewielki rozmiar metod pomocniczych kompilator JIT może je skutecznie inlinować, minimalizując narzut wykonania.

Rezultat

Każda metoda pomocnicza zawiera jedynie kilka instrukcji, dzięki czemu pozostaje poniżej limitu rozmiaru metody w JVM. System jest obecnie w stanie obsługiwać gramatyki z setkami stanów LR(1) bez naruszenia ograniczeń platformy.

6.6.2. Transmisja referencji między etapami kompilacji

Problem

Akcje semantyczne w definicji parsera mogą odwoływać się do kontekstu parsera oraz zmiennych z otaczającego zakresu leksykalnego. Podczas generacji kodu w makrze referencje te muszą zostać przepisane w taki sposób, aby odnosiły się do odpowiednich symboli w wygenerowanym kodzie.

Przyczyna

Makra operują na reprezentacji drzewa składni abstrakcyjnej (AST). Zmienne z oryginalnego zakresu leksykalnego nie istnieją w kontekście wygenerowanej klasy. Bezpośrednie kopiowanie referencji prowadziło do błędów typu „symbol not found”.

Rozwiązanie

Zaimplementowano klasę `ReplaceRefs` realizującą transformację AST poprzez zmianę przypisania symboli (ang. *re-owning*). Transformacja przebiega według następujących kroków:

1. Analiza AST kodu akcji semantycznej.
2. Identyfikacja referencji do starego kontekstu (np. parametru makra `ctx`).
3. Zastąpienie ich referencjami do nowego kontekstu (parametr metody w wygenerowanej klasie).
4. Zachowanie pozostałych referencji bez zmian.

Klasa `ReplaceRefs` rozszerza `TreeMap` z API refleksji TASTy, co umożliwia rekurencyjne przejście po całym AST i odpowiednią transformację symboli.

Rezultat

Akcje semantyczne prawidłowo odwołują się do kontekstu parsera oraz zmiennych otaczających, bez generowania błędów typowania.

6.6.3. Typy rafinowane a wsparcie IDE

Problem

W wersji pozbawionej typów rafinowanych interfejs API nie dostarczał środowisku IDE informacji o dostępnych tokenach. Funkcje autouzupełniania, podpowiedzi typów oraz mechanizmy *go-to-definition* działały w sposób ograniczony.

Przyczyna

System typów musiał dysponować informacją o polach dostępnych w wartości zwracanej przez lekser, przy czym informacja ta była pierwotnie dostępna jedynie na poziomie makra w fazie kompilacji.

Rozwiązanie

Zaimplementowano system typów rafinowanych, w którym:

- każdy token reprezentowany jest jako pole w typie rafinowanym,
- typ rafinowany udostępnia `type Fields` definiujący `NamedTuple` ze wszystkimi tokenami,
- środowisko IDE ma dostęp do pełnych informacji typów i może oferować wsparcie w postaci autouzupełniania oraz podpowiedzi,
- dostęp do pola tokena (np. `c.NUMBER`) jest statycznie typowany i bezpieczny.

Rezultat

Osiągnięto pełną integrację z IDE (Metals), obejmującą autouzupełnianie, podpowiedzi typów oraz wczesne zgłaszanie błędów typów.

Rysunek 6.2. Wsparcie IDE dzięki typom rafinowanym w projekcie ALPACA.

6.6.4. Wydajność kompilacji makr

Problem

Dla złożonych gramatyk generacja tabel LR(1) w makrze była czasochłonna, co wydłużało czas kompilacji projektu użytkownika.

Przyczyna

Algorytmy *Closure* i *Goto* dla LR(1) wymagają wielu iteracji oraz porównań zbiorów stanów. Obliczenia wykonywane na reprezentacji AST w makrze generują dodatkowy narzut.

Rozwiązanie

- Zoptymalizowano algorytmy LR(1), zastępując naiwne pętle bardziej efektywnymi strukturami danych (np. buforami modyfikowalnymi zamiast list).
- Wprowadzono buforowanie wyników pośrednich (m.in. zbiorów *FIRST*).
- Zastosowano leniwą ewaluację — niektóre stany LR(1) generowane są wyłącznie wtedy, gdy są rzeczywiście osiągalne.

Rezultat

Czas kompilacji złożonych gramatyk utrzymuje się poniżej 10 sekund (w zależności od rozmiaru gramatyki), co jest akceptowalne z perspektywy użytkownika biblioteki.

6.6.5. Testowanie makr i metaprogramowania

Problem

Testowanie systemów opartych na makrach kompilacyjnych jest utrudnione, ponieważ makra wykonują się na etapie kompilacji, a błędy często ujawniają się dopiero w wygenerowanym kodzie.

Przyczyna

Tradycyjne frameworki testowe (np. munit, ScalaTest) operują na kodzie uruchamianym w fazie wykonania, podczas gdy makra wymagają weryfikacji zarówno poprawności generowanego kodu, jak i samych efektów kompilacji.

Rozwiązanie

- Zdefiniowano testy pozytywne, sprawdzające, czy poprawne dane wejściowe generują kod, który się kompiluje i działa zgodnie z oczekiwaniami.
- Przygotowano testy negatywne, sprawdzające, czy błędne definicje są odrzucane z czytelnymi komunikatami błędów.
- Zaimplementowano testy wydajnościowe (benchmarki) dla wybranych przypadków (parser wyrażeń arytmetycznych, parser formatu JSON).
- Opracowano testy integracyjne typu end-to-end, obejmujące pełen przepływ (lekser + parser).

Każdy test makra weryfikuje zarówno poprawność kompilacji, jak i semantykę wygenerowanego kodu.

6.6.6. Błędy w kolejności definicji tokenów

Problem

Topologia definiowania tokenów może prowadzić do błędów leksykalnych, jeśli krótsze wzorce zostały umieszczone przed dłuższymi wariantami zawierającymi je jako prefiks.

Przykład: użytkownik może zdefiniować operatory porównania w naturalnej kolejności:

```
LT = <
LE = ⩽
GT = >
GE = ⩾
```

Notacja jest składniowo poprawna, lecz prowadzi do błędów w fazie parsowania wyrażenia `x ⩽ 5`, gdyż system rozpozna jedynie `<`, podczas gdy `=` zostanie odrzucony jako symbol nieznany, co spowoduje błąd składniowy w kolejnych fazach analizy.

Przyczyna

Biblioteczne wyrażenia regularne determinują, że tokenizacja przebiega sekwencyjnie według porządku definicji, bez zastosowania heurystyki maksymalnego dopasowania (zob. rozdział 4.2).
Trudność w diagnozowaniu tego typu błędów wynika z następujących czynników:

- Błąd nie jest sygnalizowany w fazie kompilacji leksera.
- Błędy ujawniają się dopiero w fazie parsowania konkretnego wejścia.
- Komunikat diagnostyczny wskazuje na parser („nieoczekiwany symbol”), a nie lekser, utrudniając identyfikację pierwotnej przyczyny.

Rozwiązanie

Konflikty takie są automatycznie wykrywane w fazie kompilacji przez dedykowany moduł `RegexChecker` (por. rozdział 4.3.6). Przed generacją kodu leksera moduł `RegexChecker` analizuje wzorce w poszukiwaniu potencjalnych konfliktów. W przypadku wykrycia nieprawidłowości użytkownik otrzymuje komunikat błędu wskazujący źródło problemu.

Rysunek 6.3. Przykład komunikatu o wykryciu konfliktu w kolejności definicji tokenów.

Rezultat

Konflikty w kolejności wzorców są wykrywane w fazie kompilacji poprzez komunikaty, które jasno wskazują, które wzorce się nakładają i wymagają zdefiniowania precedencji.

Korekta wymaga zmiany kolejności definicji tokenów (np. umieszczenie `<=` przed `<`) lub modyfikacji wyrażeń regularnych, aby wyeliminować nakładania się wzorców.

Wczesna detekcja konfliktów wzorców okazała się istotna przy implementacji przykładowych parserów (parser wyrażeń arytmetycznych, parser JSON).

6.6.7. Konflikty precedencji operatorów

Problem

Definiowanie gramatyk dla wyrażeń arytmetycznych wymaga określenia reguł precedencji operatorów, aby wymusić poprawny porządek wykonywania operacji (np. mnożenie przed dodawaniem).

Brak jawnych reguł precedencji powoduje wieloznaczność strukturalną wyrażenia `2 + 3 * 4`, które może być interpretowane jako $(2+3)\times 4 = 20$ lub $2+(3\times 4) = 14$. System sygnalizuje błąd kompilacji wskazujący na konflikt typu shift–reduce, który wymaga jawnego rozstrzygnięcia przez użytkownika.

Błędy mogą wynikać również z niespójności w definicjach reguł precedencji. Na przykład, próba definiowania operatora unarnego minus przy użyciu zwykłego operatora odejmowania może prowadzić do cyklicznych zależności precedencji, gdzie $-$ ma pierwszeństwo przed $*$, a $*$ przed $-$, co jest niemożliwe do rozstrzygnięcia.

Przyczyna

Wiele naturalnych gramatyk języków programowania zawiera niejednoznaczności, takie jak wyrażenia arytmetyczne z różnymi poziomami precedencji operatorów, czy konstrukcje warunkowe z „dangling else":

Bez takiego mechanizmu użytkownik byłby zmuszony do przepisania gramatyki do formy jednoznacznej (np. przez wprowadzenie odrębnych nieterminali dla każdego poziomu precedencji), co znacznie komplikuje specyfikację i zmniejsza jej przejrzystość.

Rozwiązanie

System ALPACA oferuje mechanizm deklaratywnego rozwiązywania konfliktów bez konieczności przepisywania gramatyki (algorytmy opisane w rozdziale 5.3.3).

Dla gramatyki wyrażeń arytmetycznych użytkownik definiuje jedynie łańcuch relacji:

```
pow before mul
mul before add
```

Notacja `X before Y` oznacza, że operator `X` ma wyższy priorytet niż operator `Y`. System automatycznie:

- wyprowadza przechodniość: Skoro `pow` przed `mul` i `mul` przed `add`, to `pow` przed `add`,
- rozwiązuje wszystkie konflikty: Używając grafu relacji pierwszeństwa,
- weryfikuje spójność: Sprawdza czy nie ma cykli w relacjach.

Jeśli podczas przetwarzania zdefiniowanych reguł system wykryje cykl (np. $A \prec B \prec C \prec A$), wyświetlana jest diagnostyka:

```
1 Inconsistent conflict resolution detected:
2 Reduction(sub) before Shift(*) before Reduction(sub)
3 There are elements being both before and after Reduction(sub)
4 at the same time.
5 Consider revising the before/after rules to eliminate cycles
```

Listing 6.1. Przykładowy komunikat o cyklu.

Komunikat wskazuje dokładne położenie sprzeczności, umożliwiając szybką korektę.

Rezultat

Zastosowanie mechanizmu rozwiązywania konfliktów przynosi kilka istotnych korzyści. Po pierwsze, pozwala na definiowanie gramatyki w formie deklaratywnej, bez konieczności ręcznego przekształcania struktury. Po drugie, znacznie redukuje liczbę wymaganych deklaracji — w praktyce z $O(n^2)$ do $O(n)$ dla typowych gramatyk. Ponadto, system wykrywa niespójności już w fazie kompilacji, a komunikaty diagnostyczne jasno wskazują źródło konfliktu, ułatwiając szybką korektę.

Zastosowanie tego mechanizmu pozwoliło na zrealizowanie kompletnego parsera wyrażeń arytmetycznych obejmującego operatory `+`, `-`, `*`, `/`, `^` oraz zagnieżdżone nawiasy w mniej niż 40 liniach kodu.

6.7. Wdrożenia, testy i eksperymenty

6.7.1. Testowanie jednostkowe

Projekt zawiera rozbudowany zestaw testów jednostkowych, obejmujących poszczególne moduły systemu.

Moduł / Komponent	**Zakres testów jednostkowych**
Lekser	Testowanie definicji tokenów (nazwy, wzorce, wartości domyślne).
	Testowanie ignorowanych reguł leksykalnych.
	Testowanie wykrywania konfliktów nazw tokenów (błędy kompilacji).
	Testowanie obsługi niepoprawnych wyrażeń regularnych.
Parser	Testowanie ekstrakcji produkcji z definicji.
	Testowanie algorytmów LR(1) (*closure*, *goto*, obliczanie zbiorów *FIRST*).
	Testowanie rozwiązywania konfliktów typu *shift–reduce*.
	Testowanie akcji semantycznych z różnymi typami kontekstu.
Funkcje pomocnicze	Testowanie derywacji (Empty[T], Copyable[T]).
	Testowanie mechanizmu withDefault[T].

Tabela 6.3. Zakres testów jednostkowych dla głównych komponentów systemu

6.7.2. Testowanie integracyjne

Projekt zawiera kilka pełnych przykładów integracyjnych, demonstrujących działanie systemu w praktyce.

Kalkulator wyrażeń arytmetycznych

Zaimplementowano kompleksowy parser wyrażeń arytmetycznych demonstrujący pełne możliwości systemu ALPACA. Parser obsługuje szeroką gamę operatorów (dodawanie, odejmowanie, mnożenie, dzielenie, modulo, potęgowanie, dzielenie całkowite) oraz funkcje matematyczne (trygonometryczne, hiperboliczne i odwrotne), a także stałe matematyczne (pi, e, tau, nieskończoność, NaN). Akcje semantyczne obliczają wartość wyrażenia poprzez wyodrębnianie wartości liczbowych z tokenów i wykonywanie odpowiednich operacji arytmetycznych.

Lekser systemu obsługuje ignorowanie białych znaków i komentarzy, definiuje operatory wieloznakowe (`**`, `//`) przed operatorami jednoznakowymi, by uniknąć konfliktów w kolejności dopasowania. Ponadto rozpoznaje liczby zmiennoprzecinkowe i całkowite (w tym notację wykładniczą) oraz funkcje matematyczne jako słowa kluczowe.

Parser definiuje kompleksowy zbiór reguł precedencji, gdzie potęgowanie ma najwyższy priorytet, następnie operatory unarny plus i minus, potem mnożenie i operatory pochodne (modulo, dzielenie całkowite), a wreszcie dodawanie i odejmowanie jako operatory o najniższym priorytecie. Ta hierarchia jest jawnie zdeklarowana w zbiorze rozwiązań konfliktów (ang. *resolutions*), zapewniając poprawną interpretację wyrażeń.
System weryfikowany był na trzech poziomach złożoności:

- Test podstawowy — proste wyrażenie $1+2$ zwracające oczekiwany wynik 3.0;
- Test złożoności średniej — wielopoziomowe nawiasowanie i operacje łączące operatory o różnych priorytetach, takie jak wyrażenie $((12+7)\cdot(3-8/(4+2))+(15-(9-3\cdot(2+1)))/5)\ldots$, potwierdzające poprawną ewaluację zagnieżdżonych wyrażeń;
- Test zaawansowany — wyrażenie zawierające funkcje trygonometryczne, operatory potęgowania i dzielenia całkowitego, takie jak $\sin(\pi/6)+\cos(\pi/3)+(2^{3^2})/(3+1)+\ldots$, demonstrujące prawidłową obsługę złożonych operacji i funkcji matematycznych.

Wszystkie testy przechodzą pomyślnie, co świadczy o poprawności implementacji leksera, parsera, zarządzania priorytetami operatorów oraz akcji semantycznych w kontekście wyrażeń o dużej złożoności strukturalnej.

Parser JSON (uproszczony)

Zaimplementowano parser dla podzbioru formatu JSON, demonstrujący obsługę złożonych struktur danych zbudowanych z elementów primitywnych. Parser obsługuje wszystkie podstawowe typy danych JSON: wartości logiczne, wartości puste (`null`), liczby zmiennoprzecinkowe, napisy, obiekty (mapy klucz-wartość) oraz tablice (listy elementów). Akcje semantyczne transformują tokeny w struktury danych reprezentujące obiekty i tablice jako typy Scali (`Map[String, Any]` oraz `List[Any]`).

Lekser ignoruje białe znaki i rozpoznaje znaki interpunkcji (`{`, `}`, `[`, `]`, `:`, `,`) jako oddzielne tokeny. Słowa kluczowe `true`, `false` i `null` są tokenizowane odpowiednio jako wartości logiczne i puste. Liczby są rozpoznawane za pośrednictwem wyrażenia regularnego obsługującego notację dziesiętną ze znakami i częściami ułamkowymi, natomiast napisy są wyodrębnianie poprzez dopasowanie tekstu w cudzysłowach (z obsługą znaków specjalnych poprzez backslash).

Parser definiuje hierarchię reguł produkcji: ‘Value‘ stanowi punkt wejścia i rozpoznaje wszystkie typy wartości JSON, zaś ‘Object‘ i Array‘ są regułami zagnieżdżonymi obsługującymi struktury złożone. Reguła ‘ObjectMembers‘ obsługuje sekwencyjne oddzielone przecinkami pary klucz-wartość, natomiast ‘ArrayElements‘ obsługuje listy elementów oddzielane przecinkami. Rekurencja lewostronna w regułach ‘ObjectMembers‘ i ArrayElements‘ pozwala na parsowanie dowolnie długich struktur, zaś parsowanie jest jednoznaczne dzięki determinizmowi składni JSON.
System weryfikowany był na czterech poziomach złożoności:

- Test primitywnych wartości — proste wyrażenie `true` zwracające wartość logiczną;
- Test struktur zagnieżdżonych — złożony obiekt JSON zawierający pola prymitywne oraz zagnieżdżone obiekty i tablice, takie jak struktury zawierające dane osobowe (imię, wiek, kursy, adres z ulicą i kodem pocztowym), demonstrujące prawidłową rekonstrukcję głębokich struktur danych;

- Test tablic obiektów — tablica zawierająca wiele obiektów o jednolitej strukturze, taka jak lista rekordów z polami identyfikacyjnymi i nazwami, potwierdzająca obsługę iteracyjnych struktur;
- Test rzeczywistej złożoności — struktura naśladująca rzeczywisty format konfiguracyjny (menu z zagnieżdżonymi polami popup i listami elementów menu z akcjami), zawierająca trzy poziomy zagnieżdżenia i mieszaniny obiektów oraz tablic.

Wszystkie testy przechodzą pomyślnie, potwierdzając poprawność implementacji leksera w obsłudze białych znaków i znaków specjalnych, prawidłowość parsowania struktur rekurencyjnych, prawidłową transformację danych przez akcje semantyczne oraz odporność systemu na wyrażenia JSON o arbitralnej złożoności strukturalnej.

6.7.3. Benchmarki wydajności

Przeprowadzono benchmarki porównujące wydajność wygenerowanego parsera *ALPACA* z innymi podejściami.

Testy wydajnościowe obejmowały dwa rodzaje gramatyk (wyrażenia arytmetyczne oraz format JSON), dla których przygotowano zarówno dane o strukturze iteracyjnej (płaskiej), jak i rekurencyjnej (głęboko zagnieżdżonej). Pomiary wykonano dla różnych rozmiarów danych wejściowych: 100, 500, 1000 oraz 2000 elementów.

System *ALPACA* porównano z biblioteką SLY (Python) reprezentującą podejście oparte na refleksji oraz z FastParse (Scala) reprezentującą kombinatory parserów. Dla każdego narzędzia mierzono czas leksykalizacji, czas parsowania oraz całkowity czas przetwarzania.

Szczegółowe wyniki testów wydajnościowych, metodologia badań oraz analiza porównawcza przedstawione są w rozdziale 7.

6.7.4. Walidacja poprawności

Walidacja gramatyk odbywała się w trzech etapach:

- Testy weryfikujące poprawną detekcję konfliktów LR(1).
- Testy sprawdzające jakość komunikatów błędów w przypadku nierozwiązanych konfliktów.
- Testy potwierdzające obsługę lewostronnej rekurencji przez parser LR(1).

Walidacja typów odbywała się w trzech etapach:

- Sprawdzanie, czy typy rafinowane poprawnie odwzorowują zdefiniowane tokeny.
- Sprawdzanie poprawności typów akcji semantycznych w kontekście wartości wydobywanych z tokenów.
- Testy negatywne, weryfikujące odrzucanie niezgodnych typowo definicji przez system typów.

Rozdział 7

Analiza porównawcza z istniejącymi rozwiązaniami

7.1. Wprowadzenie do badań porównawczych

Weryfikacja empiryczna tezy pracy wymaga systematycznego porównania systemu *ALPACA* z reprezentatywnymi rozwiązaniami istniejącymi na rynku. W tym celu przeprowadzono serię testów wydajnościowych (ang. *benchmarks*), których metodologia, wyniki oraz interpretacja zostaną przedstawione w niniejszym rozdziale.

Wybór narzędzi porównawczych wynikał z analizy przeprowadzonej w sekcji 1.4. Jako rozwiązania reprezentatywne dla głównych kategorii wybrano: SLY [11] — bibliotekę dla języka Python, reprezentującą podejście oparte na refleksji i dynamicznym typowaniu (kategoria bibliotek w językach interpretowanych) oraz FastParse [18] — bibliotekę kombinatorów parserów dla Scali, reprezentującą nowoczesne podejście do budowy parserów w językach statycznie typowanych (kategoria kombinatorów parserów).

Wybór tych narzędzi uzasadniony jest ich reprezentatywnością dla różnych paradygmatów konstrukcji parserów oraz dostępnością dokumentacji i aktywnym wsparciem społeczności. Pominięto generatory kodu takie jak ANTLR ze względu na różnice w architekturze (zewnętrzny krok generacji vs. metaprogramowanie w czasie kompilacji), które utrudniają bezpośrednie porównanie wydajności.

7.2. Metodologia badań

7.2.1. Środowisko testowe

Wszystkie testy zostały przeprowadzone w kontrolowanym środowisku o następujących parametrach.

Przed każdym pomiarem wykonywano fazę rozgrzewki (ang. *warmup*) składającą się z trzech iteracji, co pozwalało na optymalizację kodu przez kompilator JIT [41] oraz stabilizację pamięci podręcznej procesora. Każdy test był powtarzany 10 razy, a wyniki uśredniano w celu redukcji wpływu losowych wahań wydajności.

Parametr	Wartość
Procesor	Apple M4 Pro, 14 rdzeni
Pamięć RAM	48 GB
System operacyjny	macOS Sequoia 15.6
JVM	OpenJDK 21.0.8
Python	3.13.5
Scala	3.8.0-RC1

Tabela 7.1. Parametry środowiska wykorzystanego do przeprowadzenia testów wydajnościowych.

7.2.2. Charakterystyka danych testowych

Testy wydajnościowe przeprowadzono na dwóch rodzajach gramatyk: prostej gramatyce wyrażeń arytmetycznych oraz gramatyce formatu JSON. Dla każdej gramatyki przygotowano dwa typy danych wejściowych różniące się strukturą.

Dane iteracyjne

Dane iteracyjne charakteryzują się płaską strukturą składniową bez głębokiego zagnieżdżenia. Dla wyrażeń arytmetycznych przyjęto formę ciągu operacji.

```
1 + 1 - 1 +
1 + 1 - 1 +
...
1 + 1 - 1
```

Listing 7.1. Struktura iteracyjnych danych testowych dla wyrażeń arytmetycznych.

Dla formatu JSON dane iteracyjne reprezentują tablicę obiektów o stałej głębokości zagnieżdżenia.

```
{
  "items": [
    { "id": 0, "name": "item_0", "value": 0, "active": true },
    { "id": 1, "name": "item_1", "value": 10, "active": false },
    ...
  ]
}
```

Listing 7.2. Struktura iteracyjnych danych testowych dla JSON.

Dane rekurencyjne

Dane rekurencyjne charakteryzują się głębokim zagnieżdżeniem struktur składniowych, co stanowi wyzwanie dla parserów ze względu na wykorzystanie stosu wywołań. Dla wyrażeń arytmetycznych przyjęto formę zagnieżdżonych nawiasów.

```
1 + 1 + (
1 + 1 + (
...
0
) - 1
) - 1
```

Listing 7.3. Struktura rekurencyjnych danych testowych dla wyrażeń arytmetycznych.

Dla formatu JSON dane rekurencyjne reprezentują głęboko zagnieżdżone obiekty.

```
{
  "id": 0,
  "name": "obj0",
  "child": {
    "id": 1,
    "name": "obj1",
    "child": {
      ...
    }
  }
}
```

Listing 7.4. Struktura rekurencyjnych danych testowych dla JSON.

7.2.3. Scenariusze testowe

Pomiary przeprowadzono dla rozmiarów danych wejściowych: 100, 500, 1000 oraz 2000 elementów (linii lub obiektów w przypadku JSON).
Dla każdego rozmiaru mierzono następujące metryki:

- Czas leksykalizacji (ang. *lex time*) — czas przekształcenia tekstu wejściowego w strumień tokenów (dotyczy ALPACA i SLY, które wyróżniają fazę leksykalizacji),
- Czas parsowania (ang. *parse time*) — czas analizy składniowej strumienia tokenów,
- Całkowity czas przetwarzania (ang. *full parse time*) — łączny czas leksykalizacji i parsowania.

FastParse realizuje leksykalizację i parsowanie w jednym przebiegu, dlatego dla tej biblioteki mierzono wyłącznie całkowity czas przetwarzania.

7.3. Implementacja parserów testowych

W celu zapewnienia porównywalności wyników, dla każdego narzędzia zaimplementowano funkcjonalnie równoważne parsery wyrażeń arytmetycznych oraz formatu JSON. Wszystkie implementacje realizują bezpośrednią ewaluację (obliczenie wartości wyrażenia) zamiast konstrukcji drzewa AST, co eliminuje różnice wydajnościowe wynikające ze strategii alokacji pamięci.

7.3.1. Implementacja w ALPACA

Parser wyrażeń arytmetycznych w systemie *ALPACA* wykorzystuje deklaratywny interfejs DSL oparty na dopasowaniu wzorców.

```
val MathLexer = lexer:
  case "\\s+" => Token.Ignored
  case operator @ ("\\+" | "-" | "\\*" | "/" | "\\(" | "\\)") =>
    Token[operator.type]
  case num @ "\\d+" => Token["num"](num.toInt)

object MathParser extends Parser:
  val Expr: Rule[Int] = rule(
    { case (Expr(a), MathLexer.`\\*`(_), Expr(b)) => a * b }: @name("mul"),
    { case (Expr(a), MathLexer.`/`(_), Expr(b)) => a / b }: @name("div"),
    { case (Expr(a), MathLexer.`\\+`(_), Expr(b)) => a + b }:
   @name("plus"),
    { case (Expr(a), MathLexer.`-`(_), Expr(b)) => a - b }: @name("minus"),
    { case (MathLexer.`\\(`(_), Expr(a), MathLexer.`\\)`(_)) => a },
    { case MathLexer.num(n) => n.value },
  )
  // Parser entry point
  val root: Rule[Int] = rule { case Expr(v) => v }

  override val resolutions = Set(
    // addition and subtraction only after multiplication and division
    Production.ofName("plus").after(MathLexer.`\\*`, MathLexer.`/`),
    Production.ofName("minus").after(MathLexer.`\\*`, MathLexer.`/`),

    // all operators are left associative (reduce before shifting)
    Production.ofName("mul").before(MathLexer.`\\*`, MathLexer.`/`),
    Production.ofName("div").before(MathLexer.`\\*`, MathLexer.`/`),
    Production.ofName("plus").before(MathLexer.`\\+`, MathLexer.`-`),
    Production.ofName("minus").before(MathLexer.`\\+`, MathLexer.`-`),
  )
```

Listing 7.5. Parser wyrażeń arytmetycznych w systemie ALPACA.

7.3.2. Implementacja w SLY

Odpowiadająca implementacja w bibliotece SLY wykorzystuje dekoratory i refleksję nazw metod.

```
class MathLexer(Lexer):
    literals = {'+', '-', '*', '/', '(', ')'}
    tokens = {NUM}
    ignore = ' \t\n'
    NUM = r'\d+'

class MathParser(Parser):
    tokens = MathLexer.tokens
    precedence = (
        ('left', '+', '-'),
        ('left', '*', '/'),
    )

    @_('expr "*" expr')
    def expr(self, p):
        return p.expr0 * p.expr1

    @_('expr "/" expr')
    def expr(self, p):
        return p.expr0 / p.expr1

    @_('expr "+" expr')
    def expr(self, p):
        return p.expr0 + p.expr1

    @_('expr "-" expr')
    def expr(self, p):
        return p.expr0 - p.expr1

    @_('"(" expr ")"')
    def expr(self, p):
        return p.expr

    @_('NUM')
    def expr(self, p):
        return int(p.NUM)
```

Listing 7.6. Parser wyrażeń arytmetycznych w bibliotece SLY.

7.3.3. Implementacja w FastParse

FastParse realizuje parsowanie poprzez kompozycję funkcji parserowych.

```
object MathParser extends Parser[Int]:
  def eval(tree: (Int, Seq[(String, Int)])) =
    val (base, ops) = tree
    ops.foldLeft(base):
      case (left, ("+", right)) => left + right
      case (left, ("-", right)) => left - right
      case (left, ("*", right)) => left * right
      case (left, ("/", right)) => left / right


  def number[$: P]: P[Int] = P(CharIn("0-9").rep(1).!.map(_.toInt))
  def parens[$: P]: P[Int] = P("(" ~/ addSub ~ ")")
  def factor[$: P]: P[Int] = P(number | parens)

  def divMul[$: P]: P[Int] =
    P(factor ~ (CharIn("*/").! ~/ factor).rep).map(eval)

  def addSub[$: P]: P[Int] =
    P(divMul ~ (CharIn("+\\-").! ~/ divMul).rep).map(eval)

  def expr[$: P]: P[Int] = P(addSub ~ End)

  def parse(input: String): Either[String, Int] =
    fastparse.parse(input, expr(using _)) match
      case Parsed.Success(value, _) => Right(value)
      case f: Parsed.Failure => Left(f.msg)
```

Listing 7.7. Parser wyrażeń arytmetycznych w bibliotece FastParse.

Charakterystyczną cechą FastParse jest brak wydzielonej fazy leksykalizacji — tokenizacja i parsowanie są realizowane w jednym przebiegu poprzez kompozycję parserów elementarnych.

7.4. Wyniki badań

7.4.1. Wyrażenia arytmetyczne — dane iteracyjne

Tabela 7.2 przedstawia wyniki testów wydajnościowych dla wyrażeń arytmetycznych o strukturze iteracyjnej.

Rozmiar	ALPACA			SLY			FastParse
	Lex	Parse	Razem	Lex	Parse	Razem	Razem
100	16,85 ms	8,49 ms	16,48 ms	1,52 ms	6,69 ms	8,24 ms	6,13 ms
500	41,94 ms	17,63 ms	30,07 ms	7,98 ms	31,18 ms	38,57 ms	14,13 ms
1000	40,58 ms	10,95 ms	77,24 ms	14,84 ms	59,65 ms	76,50 ms	21,24 ms
2000	166,89 ms	21,22 ms	188,56 ms	30,61 ms	124,52 ms	156,70 ms	7,73 ms

Tabela 7.2. Wyniki testów wydajnościowych dla wyrażeń arytmetycznych (dane iteracyjne).

7.4.2. Wyrażenia arytmetyczne — dane rekurencyjne

Tabela 7.3 przedstawia wyniki dla wyrażeń arytmetycznych o strukturze rekurencyjnej (głębokie zagnieżdżenie nawiasów).

Rozmiar	**ALPACA**			**SLY**			**FastParse**
	Lex	Parse	Razem	Lex	Parse	Razem	Razem
100	3,55 ms	1,70 ms	4,96 ms	1,79 ms	7,27 ms	9,25 ms	3,10 ms
500	27,70 ms	7,81 ms	29,92 ms	10,45 ms	35,30 ms	46,85 ms	4,56 ms
1000	63,79 ms	12,94 ms	80,08 ms	20,22 ms	72,95 ms	96,95 ms	7,29 ms
2000	220,96 ms	25,82 ms	274,95 ms	43,06 ms	150,68 ms	194,77 ms	†

Tabela 7.3. Wyniki testów wydajnościowych dla wyrażeń arytmetycznych (dane rekurencyjne). † oznacza błąd przepełnienia stosu (*StackOverflowError*).

Wyniki ujawniają istotne ograniczenie biblioteki FastParse: dla danych rekurencyjnych o głębokości przekraczającej 1000 poziomów zagnieżdżenia występuje błąd przepełnienia stosu (ang. *StackOverflowError*). Jest to konsekwencją architektury kombinatorów parserów, które realizują rekurencję poprzez stos wywołań JVM.

7.4.3. Format JSON — dane iteracyjne

Tabela 7.4 przedstawia wyniki dla formatu JSON o strukturze iteracyjnej (tablica obiektów).

Rozmiar	**ALPACA**			**SLY**			**FastParse**
	Lex	Parse	Razem	Lex	Parse	Razem	Razem
100	30,63 ms	4,09 ms	26,27 ms	7,22 ms	17,07 ms	24,88 ms	15,82 ms
500	144,00 ms	20,55 ms	164,02 ms	35,60 ms	87,15 ms	131,62 ms	11,67 ms
1000	478,19 ms	46,43 ms	555,45 ms	75,34 ms	182,93 ms	263,86 ms	12,64 ms
2000	1,72 s	117,50 ms	1,58 s	151,88 ms	374,71 ms	550,08 ms	16,58 ms

Tabela 7.4. Wyniki testów wydajnościowych dla formatu JSON (dane iteracyjne).

7.4.4. Format JSON — dane rekurencyjne

Tabela 7.5 przedstawia wyniki dla formatu JSON o strukturze rekurencyjnej (głęboko zagnieżdżone obiekty).

Wyniki dla rekurencyjnych danych JSON ujawniają problem wydajnościowy w module leksykalizacji systemu *ALPACA* dla danych o wysokim stopniu zagnieżdżenia. Czas leksykalizacji rośnie nieproporcjonalnie do rozmiaru danych, co wymaga dalszej analizy i optymalizacji (sekcja 7.5.3).

Rozmiar	ALPACA			SLY			FastParse
	Lex	Parse	Razem	Lex	Parse	Razem	Razem
100	44,96 ms	4,33 ms	49,44 ms	20,33 ms	17,77 ms	39,81 ms	3,47 ms
500	2,99 s	18,09 ms	3,02 s	358,04 ms	95,21 ms	459,70 ms	15,19 ms
1000	23,55 s	37,51 ms	23,52 s	1,36 s	189,58 ms	1,58 s	86,38 ms
2000	*	*	*	5,26 s	385,47 ms	5,79 s	346,44 ms

Tabela 7.5. Wyniki testów wydajnościowych dla formatu JSON (dane rekurencyjne).
* oznacza, że system nie zakończył pomiaru w akceptowalnym czasie.

7.5. Analiza wyników

7.5.1. Porównanie faz przetwarzania

Analiza wyników ujawnia zróżnicowaną charakterystykę wydajnościową poszczególnych faz przetwarzania.

Faza leksykalizacji

Lekser biblioteki SLY wykazuje stabilną i przewidywalną wydajność, charakteryzującą się liniową złożonością względem rozmiaru danych wejściowych. Lekser systemu *ALPACA* osiąga porównywalną wydajność dla danych iteracyjnych, jednak wykazuje znaczący spadek wydajności dla danych rekurencyjnych JSON.

Przyczyną tego zachowania jest sposób obsługi białych znaków w obu systemach. W danych testowych JSON każdy poziom zagnieżdżenia dodaje dwie spacje wcięcia. Dla testu `recursive_json_2000` najbardziej zagnieżdżony obiekt jest wcięty za pomocą 4000 spacji. System *ALPACA* tokenizuje każdą sekwencję białych znaków jako osobny token typu `Ignored`, co wymaga dopasowania wzorca wyrażenia regularnego dla każdej spacji.

Biblioteka SLY stosuje odmienne podejście — pomijanie ignorowanych znaków realizowane jest prostym warunkiem iteracyjnym.

```
# Fragment of the internal SLY implementation
# Skipping ignored characters without using regex
if text[index] in _ignore:
    index += 1
    continue
```

Listing 7.8. Mechanizm pomijania białych znaków w bibliotece SLY.

To podejście eliminuje narzut związany z dopasowaniem wyrażeń regularnych dla białych znaków, co tłumaczy znaczącą przewagę wydajnościową SLY w fazie leksykalizacji dla danych o dużym stopniu zagnieżdżenia. Optymalizacja mechanizmu ignorowania znaków stanowi kierunek przyszłych prac opisanych w sekcji 7.7.1.

Faza parsowania

Faza parsowania systemu *ALPACA* wykazuje stabilną wydajność, znacząco przewyższającą bibliotekę *SLY* dla wszystkich testowanych scenariuszy. Różnica ta wynika z różnic architektonicznych. System *ALPACA* wykorzystuje tabele parsowania LR(1) generowane

w czasie kompilacji, co eliminuje narzut interpretacji w czasie wykonania. Z kolei biblioteka *SLY* interpretuje reguły gramatyczne w czasie wykonania z wykorzystaniem refleksji Pythona, co wprowadza znaczący narzut.

7.5.2. Zachowanie przy głębokim zagnieżdżeniu

Testy z danymi rekurencyjnymi ujawniły istotne różnice w obsłudze głęboko zagnieżdżonych struktur. Biblioteka *FastParse* wykazuje przepełnienie stosu dla głębokości zagnieżdżenia przekraczającej 1000–2000 poziomów, co jest konsekwencją rekursywnej natury kombinatorów parserów. System *ALPACA* charakteryzuje się stabilnym działaniem parsera niezależnie od głębokości zagnieżdżenia dzięki wykorzystaniu automatu ze stosem zamiast rekurencji. Biblioteka *SLY* również wykazuje stabilne działanie dla wszystkich testowanych głębokości, analogicznie do systemu *ALPACA*. Obserwacja ta potwierdza przewagę parserów LR nad kombinatorami parserów w scenariuszach wymagających obsługi głęboko zagnieżdżonych struktur.

7.5.3. Wnioski

Przeprowadzone eksperymenty pozwalają na sformułowanie następujących wniosków dotyczących wydajności systemu *ALPACA*.

Zalety systemu:

- Faza parsowania systemu *ALPACA* charakteryzuje się stabilną i przewidywalną wydajnością, przewyższającą bibliotekę *SLY* we wszystkich analizowanych scenariuszach testowych.
- Zastosowanie parsera klasy LR(1) eliminuje ograniczenia wynikające z rozmiaru stosu wywołań, umożliwiając poprawne przetwarzanie struktur o dowolnej głębokości zagnieżdżenia.
- Liniowa złożoność czasowa procesu parsowania względem liczby tokenów zapewnia deterministyczne i przewidywalne czasy przetwarzania danych wejściowych.

Obszary wymagające optymalizacji

Mechanizm obsługi znaków ignorowanych wykazuje obniżoną wydajność w przypadku danych wejściowych charakteryzujących się dużym stopniem wcięcia. Wynika to z faktu, że każda sekwencja znaków białych jest tokenizowana jako osobna jednostka, co prowadzi do zwiększonego narzutu obliczeniowego.

7.6. Porównanie interfejsów programistycznych

Oprócz wydajności, istotnym kryterium oceny narzędzi jest jakość interfejsu programistycznego (API). Tabela 7.6 zestawia kluczowe aspekty interfejsów porównywanych narzędzi.

Kryterium	ALPACA	SLY	FastParse
Wydajność parsowania	wysoka	niska	wysoka
Wydajność leksykalizacji	średnia*	wysoka	nie dotyczy
Obsługa głębokiego zagnieżdżenia	pełna	pełna	ograniczona
Bezpieczeństwo typów	pełne	brak	pełne
Integracja IDE	pełna	ograniczona	pełna
Diagnostyka błędów	dobra	średnia	dobra

Tabela 7.6. Podsumowanie analizy porównawczej (* wymaga optymalizacji mechanizmu ignorowania białych znaków).

7.6.1. Bezpieczeństwo typów

System *ALPACA* oferuje pełne bezpieczeństwo typów na poziomie kompilacji dzięki wykorzystaniu typów rafinowanych (sekcja 3.1.8). Każdy token i wynik parsowania posiada precyzyjny typ znany kompilatorowi, co umożliwia wykrywanie błędów przed uruchomieniem programu.

W przeciwieństwie do tego, biblioteka SLY wykorzystuje dynamiczne typowanie Pythona, co oznacza, że błędy typów są wykrywane dopiero w czasie wykonania. Fragment kodu 7.9 ilustruje sytuację, w której błąd typowania w akcji semantycznej zostanie wykryty dopiero podczas parsowania konkretnego wejścia.

```
class Calculator(Parser):
    @_('NUM')
    def expr(self, p):
        return p.NUM

    @_('expr "+" "+"')
    def expr(self, p):
        # TypeError: can only concatenate str (not "int") to str
        # We have never casted NUM to int, so p.expr0 is still a str
        return p.expr0 + 1
```

Listing 7.9. Błąd typowania wykrywany dopiero w czasie wykonania (SLY).

7.6.2. Integracja ze środowiskiem IDE

System *ALPACA* oferuje pełną integrację ze standardowymi narzędziami IDE dla języka Scala (IntelliJ IDEA, Metals) bez konieczności instalacji dedykowanych wtyczek. Funkcjonalności takie jak: automatyczne uzupełnianie nazw tokenów, nawigacja do definicji (*go-to-definition*), prezentacja typów przy najechaniu kursorem oraz wykrywanie błędów w czasie rzeczywistym są dostępne natywnie dzięki wykorzystaniu systemu typów Scali.

Biblioteka SLY, ze względu na wykorzystanie refleksji i niestandardowych konwencji (nazwy metod jako produkcje, dekoratory `@_()`), oferuje ograniczone wsparcie IDE. Analizatory statyczne, takie jak mypy, generują liczne błędy dla poprawnego kodu SLY, co ilustruje fragment 7.10.

```
> mypy ./math_parser.py
math_parser.py:2: error: Skipping analyzing "sly": module is installed, but
    missing library stubs or py.typed marker  [import-untyped]
math_parser.py:2: note: See https://mypy.readthedocs.io/en/stable/
   running_mypy.html#missing-imports
math_parser.py:5: error: Name "PLUS" is not defined  [name-defined]
math_parser.py:5: error: Name "MIN" is not defined  [name-defined]
math_parser.py:5: error: Name "MUL" is not defined  [name-defined]
math_parser.py:5: error: Name "DIV" is not defined  [name-defined]
math_parser.py:5: error: Name "LPAREN" is not defined  [name-defined]
math_parser.py:5: error: Name "RPAREN" is not defined  [name-defined]
math_parser.py:5: error: Name "NUM" is not defined  [name-defined]
math_parser.py:25: error: Name "PLUS" is not defined  [name-defined]
math_parser.py:25: error: Name "MIN" is not defined  [name-defined]
math_parser.py:26: error: Name "MUL" is not defined  [name-defined]
math_parser.py:26: error: Name "DIV" is not defined  [name-defined]
math_parser.py:29: error: Name "_" is not defined  [name-defined]
math_parser.py:33: error: Name "_" is not defined  [name-defined]
math_parser.py:37: error: Name "expr" already defined on line 33  [no-redef
   ]
math_parser.py:37: error: Name "_" is not defined  [name-defined]
math_parser.py:41: error: Name "expr" already defined on line 33  [no-redef
   ]
math_parser.py:41: error: Name "_" is not defined  [name-defined]
math_parser.py:45: error: Name "expr" already defined on line 33  [no-redef
   ]
math_parser.py:45: error: Name "_" is not defined  [name-defined]
math_parser.py:49: error: Name "expr" already defined on line 33  [no-redef
   ]
math_parser.py:49: error: Name "_" is not defined  [name-defined]
math_parser.py:53: error: Name "expr" already defined on line 33  [no-redef
   ]
math_parser.py:53: error: Name "_" is not defined  [name-defined]
Found 24 errors in 1 file (checked 1 source file)
```

Listing 7.10. Wynik analizy statycznej mypy dla parsera SLY.

Większość zgłoszonych błędów wynika z dynamicznego charakteru biblioteki SLY: tokeny są definiowane jako zmienne klasowe bez jawnej deklaracji typu, a dekorator `@_()` nie jest rozpoznawany przez analizator statyczny. Ponadto konwencja definiowania wielu metod o tej samej nazwie (`expr`) dla alternatywnych produkcji jest traktowana jako błąd redefinicji.

Rysunek 7.1 przedstawia widok kodu SLY w środowisku VS Code, gdzie widoczne są ostrzeżenia generowane przez wbudowany analizator statyczny.

sly > math_parser.py > ...

```python
from pathlib import Path
from sly import Lexer, Parser

class MathLexer(Lexer):
    tokens: set = {PLUS, MIN, MUL, DIV, LPAREN, RPAREN, NUM}
    ignore: str = ' \t\n'

    PLUS = r'\+'
    MIN = r'-'
    MUL = r'\*'
    DIV = r'/'
    LPAREN = r'\('
    RPAREN = r'\)'
    NUM = r'\d+'

    def error(self, t) -> None:
        print(f"Illegal character '{t.value[0]}'")
        self.index += 1


class MathParser(Parser):
    tokens: set = MathLexer.tokens

    precedence: tuple[tuple[str, Any, Any],...] = (
        ('left', PLUS, MIN),
        ('left', MUL, DIV),
    )

    @_('expr')
    def program(self, p) -> Any:
        return p.expr

    @_('expr MUL expr')
    def expr(self, p) -> Any:
        return p.expr0 * p.expr1

    @_('expr DIV expr')
    def expr(self, p) -> Any:
        return p.expr0 / p.expr1

```

Rysunek 7.1. Ostrzeżenia IDE dla poprawnego kodu parsera SLY w środowisku VS Code.

7.7. Podsumowanie analizy porównawczej

Przeprowadzona analiza porównawcza pozwala na weryfikację tezy postawionej w rozdziale 1. W odniesieniu do pytań badawczych sformułowanych w sekcji 1.2, można sformułować następujące wnioski:

- Faza parsowania systemu *ALPACA* osiąga wydajność porównywalną z FastParse i znacząco przewyższa bibliotekę SLY. Moduł leksykalizacji wymaga dalszej optymalizacji dla niektórych wzorców wyrażeń regularnych.
- Interfejs API oparty na dopasowaniu wzorców Scali oferuje naturalne wyrażenie reguł gramatycznych przy zachowaniu pełnego bezpieczeństwa typów i integracji z IDE.
- System generuje komunikaty błędów zawierające kontekst syntaktyczny.

7.7.1. Kierunki dalszych prac

Na podstawie przeprowadzonych badań zidentyfikowano następujące kierunki dalszego rozwoju systemu *ALPACA*:

- Wprowadzenie dedykowanego mechanizmu pomijania białych znaków, analogicznego do rozwiązania stosowanego w bibliotece SLY, który eliminowałby narzut tokenizacji dla sekwencji spacji i tabuladorów.
- Implementacja algorytmu LALR(1) w celu redukcji rozmiaru tabel parsowania dla dużych gramatyk.
- Przeprowadzenie testów na większej liczbie gramatyk reprezentujących rzeczywiste języki programowania (np. podzbiór Scali, SQL).

Bibliografia

[1] A. V. Aho, M. S. Lam, R. Sethi i J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2 wyd. Addison-Wesley, 2006.

[2] J. E. Hopcroft, R. Motwani i J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3 wyd. Addison-Wesley, 2006.

[3] N. Chomsky. „Three models for the description of language". W: *IRE Transactions on Information Theory* 2.3 (1956), s. 113–124.

[4] K. Thompson. „Programming Techniques: Regular expression search algorithm". W: *Communications of the ACM* 11.6 (1968), s. 419–422.

[5] P. M. Lewis II i R. E. Stearns. „Syntax-directed transduction". W: *Journal of the ACM* 15.3 (1968), s. 465–488.

[6] D. E. Knuth. „On the translation of languages from left to right". W: *Information and Control* 8.6 (1965), s. 607–639.

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] *Dropped: Scala 2 Macros. Scala 3 Reference*. URL: https://docs.scala-lang.org/scala3/reference/metaprogramming/migrating/macro-compatibility.html (term. wiz. 25. 10. 2025).

[21] *Metaprogramming. Scala 3 Reference*. URL: https://docs.scala-lang.org/scala3/reference/metaprogramming.html (term. wiz. 25. 10. 2025).

[22] N. Stucki. *Scalable Metaprogramming in Scala 3. EPFL Infoscience page*. 2024. URL: https://infoscience.epfl.ch/entities/publication/6dd02f9b-1f9b-4c9c-9748-ddf1634c1630 (term. wiz. 25. 10. 2025).

[23] N. Stucki, A. Biboudis, S. Doeraene i M. Odersky. „Semantics-preserving inlining for metaprogramming". W: *Proceedings of the 11th ACM SIGPLAN International Symposium on Scala*. 2020. DOI: 10.1145/3426426.3428486. URL: https://dl.acm.org/doi/10.1145/3426426.3428486.

[24] N. Stucki. „Scalable Metaprogramming in Scala 3". Prac. dokt. Lausanne: EPFL, 2020.

[25] *Runtime Multi-Stage Programming. Scala 3 Reference*. URL: https://docs.scala-lang.org/scala3/reference/metaprogramming/staging.html (term. wiz. 25. 10. 2025).

[26] N. Stucki, J. Brachthäuser i M. Odersky. „A practical unification of multi-stage programming and macros". W: *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*. 2018. DOI: 10.1145/3278122.3278139. URL: https://dl.acm.org/doi/10.1145/3278122.3278139.

[27] N. Stucki, A. Biboudis i M. Odersky. „Multi-stage programming with generative and analytical macros". W: *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala*. 2021. DOI: 10.1145/3486609.3487203. URL: https://dl.acm.org/doi/10.1145/3486609.3487203.

[28] *Reflection. Scala 3 Reference*. URL: https://docs.scala-lang.org/scala3/reference/metaprogramming/reflection.html (term. wiz. 25. 10. 2025).

[29] *Quoted Code | Macros in Scala 3*. URL: https://docs.scala-lang.org/scala3/guides/macros/quotes.html (term. wiz. 25. 10. 2025).

[30] N. Stucki, A. Biboudis, S. Doeraene i M. Odersky. „Semantics-preserving inlining for metaprogramming". W: SCALA 2020 (2020), s. 14–24. DOI: 10.1145/3426426.3428486. URL: https://doi.org/10.1145/3426426.3428486.

[31] Y. Lilis i A. Savidis. „A Survey of Metaprogramming Languages". W: *ACM Comput. Surv.* 52.6 (paź. 2019). DOI: 10.1145/3354584. URL: https://doi.org/10.1145/3354584.

[32] *Macros. Scala 3 Reference*. URL: https://docs.scala-lang.org/scala3/reference/metaprogramming/macros.html (term. wiz. 25. 10. 2025).

[33] *Scala 3 Macros*. URL: https://docs.scala-lang.org/scala3/guides/macros/macros.html (term. wiz. 25. 10. 2025).

[34] *Reflection | Macros in Scala 3*. URL: https://docs.scala-lang.org/scala3/guides/macros/reflection.html (term. wiz. 25. 10. 2025).

[35] J. McCarthy. „Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". W: *Communications of the ACM* 3.4 (kw. 1960), s. 184–195. DOI: 10.1145/367177.367199.

[36] T. Sheard i S. Peyton Jones. „Template Meta-programming for Haskell". W: *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*. Haskell '02. Pittsburgh, Pennsylvania, USA: ACM, 2002, s. 1–16. DOI: 10.1145/581690.581691.

[37] Rust Team. *The Rust Programming Language: Macros*. The Rust Foundation. 2024. URL: https://doc.rust-lang.org/book/ch19-06-macros.html (term. wiz. 05. 12. 2024).

[38] S. Klabnik i C. Nichols. „The Rust Programming Language". W: San Francisco, CA, USA: No Starch Press, 2018.

[39] *Selectable | Macros in Scala 3*. URL: https://docs.scala-lang.org/scala3/reference/changed-features/structural-types.html (term. wiz. 27. 11. 2025).

[40] *Computed Field Names | Macros in Scala 3*. URL: https://docs.scala-lang.org/scala3/reference/other-new-features/named-tuples.html#:~:text=Computed%20Field%20Names (term. wiz. 27. 11. 2025).

[41] T. Lindholm, F. Yellin, G. Bracha i A. Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Java SE 8. Specyfikacja JVM definiuje limit rozmiaru kodu bajtowego metody na 65536 bajtów. Addison-Wesley Professional, 2014.

[42] B. Kozak. *Method too large*. URL: https://halotukozak.github.io/posts/scala-macro-jvm-method-size-limit/ (term. wiz. 27. 11. 2025).

[43] *Type Class Derivation | Scala 3 Reference*. URL: https://docs.scala-lang.org/scala3/reference/contextual/derivation.html (term. wiz. 29. 11. 2025).

[44] E. Burmako. „Scala Macros: Let Our Powers Combine!" W: *Proceedings of the 4th Workshop on Scala*. 2013. DOI: 10.1145/2489837.2489840. URL: https://dl.acm.org/doi/10.1145/2489837.2489840.

[45] B. C. Pierce. *Types and Programming Languages*. Cambridge, Massachusetts: MIT Press, 2002.

[46] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman i M. Zenger. *An Overview of the Scala Programming Language*. Spraw. tech. IC/2004/64. EPFL, 2004. URL: https://lampwww.epfl.ch/~odersky/papers/ScalaOverview.pdf.

[47] E. Gamma, R. Helm, R. Johnson i J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[48] *Shapeless: Generic programming for Scala*. URL: https://github.com/milessabin/shapeless (term. wiz. 29. 11. 2025).

[49] *Magnolia: Fast, easy and transparent typeclass derivation for Scala*. URL: https://github.com/softwaremill/magnolia (term. wiz. 29. 11. 2025).

[50] *Lexical Analysis*. URL: https://www.cs.cornell.edu/courses/cs4120/2022sp/notes.html?id=lexing (term. wiz. 06. 12. 2025).

[51] *Lexical analysis*. URL: https://en.wikipedia.org/wiki/Lexical_analysis#Scanner (term. wiz. 06. 12. 2025).

[52] A. Kościelski. *Języki formalne i automaty*. URL: https://ii.uni.wroc.pl/~kosciels/jf1996/jf.pdf (term. wiz. 06. 12. 2025).

[53] G. M. Arces, E. V. C. Mangaoang, M. A. J. B. Regis i J. J. A. Barrera. „Development of a Non-Deterministic Finite Automaton with Epsilon Moves (E−NFA) Generator Using Thompson's Construction Algorithm". W: *Journal of Science, Engineering and Technology* 6.1 (grud. 2018), s. 149–159. DOI: 10.61569/tvex6p34. URL: https://journals.southernleytestateu.edu.ph/index.php/jset/article/view/214.

[54] *Regex - Scala Standard Library*. URL: https://www.scala-lang.org/api/3.x/scala/util/matching/Regex.html (term. wiz. 06. 12. 2025).

[55] Marianobarrios. *dregex: Deterministic Regular Expression Engine*. Java library for regular expression algebra and subset checking. 2016. URL: https://github.com/marianobarrios/dregex (term. wiz. 13. 12. 2025).

[56] *Pushdown automaton*. URL: https://en.wikipedia.org/wiki/Pushdown_automaton (term. wiz. 06. 12. 2025).

[57] *Predictive Parsing*. URL: https://www.naukri.com/code360/library/predictive-parsing (term. wiz. 06. 12. 2025).

[58] M. Tomita i S. Ng. „The Generalized LR Parsing Algorithm". W: *Generalized LR Parsing*. Red. M. Tomita. Boston, MA: Springer, 1991, s. 1–16. DOI: 10.1007/978-1-4615-4034-2_1. URL: https://link.springer.com/chapter/10.1007/978-1-4615-4034-2_1.

[59] *Parser LL*. URL: https://pl.wikipedia.org/wiki/Parser_LL (term. wiz. 06. 12. 2025).

[60] *Parser LR*. URL: https://pl.wikipedia.org/wiki/Parser_LR (term. wiz. 06. 12. 2025).

[61] D. E. Knuth. „On the translation of languages from left to right". W: *Information and Control* 8.6 (czer. 1965), s. 607–639. DOI: 10.1016/S0019-9958(65)90426-2. URL: https://doi.org/10.1016/S0019-9958(65)90426-2.

[62] *Parsing Conflicts*. URL: https://courses.grainger.illinois.edu/cs421/sp2009/lectures/lecture10.pdf (term. wiz. 06. 12. 2025).

[63] A. Szeruda. „Narzędzie do generowania dobrze typowanych parserów w stylu przekazywania kontynuacji". Praca licencjacka. Uniwersytet Wrocławski, Instytut Informatyki, 2023. URL: https://ii.uni.wroc.pl/media/uploads/2023/11/16/szeruda-uwr-28-lic-69452-233823.pdf (term. wiz. 07. 12. 2025).

[64] A. V. Aho i S. C. Johnson. „LR Parsing". W: *ACM Computing Surveys* 6.2 (czer. 1974), s. 99–124. DOI: 10.1145/356628.356629. URL: https://dl.acm.org/doi/10.1145/356628.356629.

[65] *Parsery LR(1) - część 2*. URL: https://kompilatory.agh.edu.pl/kompilatory/wyklady/WEAIiE-08-Parsery-LR(1)-czesc-2.pdf (term. wiz. 06. 12. 2025).

[66] M. A. Bender i D. Ron. „Testing properties of directed graphs: acyclicity and connectivity". W: *Random Structures & Algorithms* (lut. 2002). DOI: 10.1002/rsa.10023. URL: https://doi.org/10.1002/rsa.10023.

[67] *ALPACA - GitHub Project*. URL: https://github.com/users/halotukozak/projects/6 (term. wiz. 09. 12. 2025).

[68] *ALPACA - Project Milestones*. URL: https://github.com/halotukozak/alpaca/milestones (term. wiz. 09. 12. 2025).

[69] *ALPACA - GitHub Repository*. URL: https://github.com/halotukozak/alpaca (term. wiz. 09. 12. 2025).

[70] *ALPACA - Documentation*. URL: https://halotukozak.github.io/alpaca/ (term. wiz. 09. 12. 2025).

Spis rysunków

Spis tabel

Spis listingów