Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Informatyki

Projekt dyplomowy

Implementacja narzędzi lex i yacc z wykorzystaniem metaprogramowania

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

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

Kraków, 2025

Spis treści

Rozdział 1

Cel 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:

1. wydajność — czas parsowania porównywalny z narzędziami opartymi na generacji kodu (ANTLR, Yacc), przewyższający biblioteki interpretowane (PLY, SLY).
2. użyteczność — interfejs programistyczny (API) niezależny od dedykowanego DSL, zintegrowany z systemem typów Scali i wspierany przez standardowe narzędzia IDE.
3. 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:

1. Czy możliwe jest osiągnięcie wydajności zbliżonej do generatorów kodu przy zachowaniu elastyczności bibliotek kombinatorów poprzez zastosowanie metaprogramowania?
2. W jakim stopniu wykorzystanie typów rafinowanych w Scali 3 wpływa na bezpieczeństwo typów i komfort pracy z wygenerowanym parserem?
3. 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:

1. 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.
2. Generowanie komunikatów błędów w oparciu o kontekst parsera LR(1).
3. 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;
}
else if ($5 == tANDOP) {
    $5 = 1;
}
```

```
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 `@_()` Dekorator ten 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 Nazwa metody 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.

Priorytet operatorów W krotce `precedence` definiowane jest pierwszeństwo operatorów, jednakże dodanie `\%~prec` pozwala nadpisać priorytet dla konkretnej reguły składniowej.

Dostęp do kontekstu 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

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].

Przykład użycia cytatów i wstawek Poniższy przykład ilustruje podstawowe wykorzystanie cytatów i wstawek w makrach:

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

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

```
7
8 // żUycie: square(3) → rozwinie ęsi do: val squared = 3 * 3; squared
```

Listing 2.1: Proste makro z wykorzystaniem cytatów i wstawek

W powyższym przykładzie:

- `'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].

Przykład naruszenia bezpieczeństwa międzyetapowego Następujący kod **nie skompiluje się**, ponieważ narusza zasady bezpieczeństwa międzyetapowego:

```
1 def unsafeQuote(using Quotes): Expr[Int] = {
2   val localVar = 42
3   '{ localVar }  // ŁĄBD: localVar nie istnieje w fazie wykonania!
4 }
```

Listing 2.2: Błąd bezpieczeństwa międzyetapowego (kod nie kompiluje się)

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`:

```
1 def safeQuote(using Quotes): Expr[Int] = {
2   val localVar = 42
3   Expr(localVar)  // OK: śćwarto jest serializowana do cytatu
4 }
```

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].

Przykład definicji inline

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

// Wywołanie: max(3, 5)
// Rozwinięcie do: if 3 > 5 then 3 else 5
// Po optymalizacji kompilatora: 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].

Przykład makra generującego kod

```
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)
}

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

Listing 2.5: Makro generujące kod inspekcji typu

W powyższym przykładzie 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.

Przykład dopasowania wzorców kodu

```
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 }  // reassocjacja
  case _ => x  // brak optymalizacji
}

// Przykład użycia:
optimize(0 + 5)      // → 5
optimize(3 * 1)      // → 3
optimize(0 * 100)    // → 0
```

Listing 2.6: 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].

Hierarchia klas refleksji TASTy 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)

Przykład użycia refleksji TASTy

```
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)
}

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

Listing 2.7: Inspekcja struktury klasy przypadku za pomocą refleksji TASTy

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.

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:

- **Lisp:** makra operują na nietypo

  wanych listach (*S-expressions*), co umożliwia dużą elastyczność, ale eliminuje sprawdzanie typów w czasie kompilacji

- **Scala 3:** makra 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. Główne podobieństwa i różnice:

- **Podobieństwa:** obie implementacje wykorzystują cytaty (`[ ... |]|` w Haskell, `'{ ... }` w Scali) oraz wstawki (`$(...)` w Haskell, `${...}` w Scali)
- **Różnice:** Template Haskell wymaga specjalnego trybu kompilacji (`-XTemplateHaskell`), podczas gdy makra Scali 3 są standardową częścią języka; 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:

- **Rust:** makra proceduralne operują na tokenach (ang. *token stream*), co daje dużą kontrolę, ale utrudnia analizę semantyczną
- **Scala 3:** makra operują na typowanych AST, co umożliwia analizę semantyczną i sprawdzanie typów wygenerowanego kodu

2.4. Zastosowania metaprogramowania w projekcie ALPACA

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

1. **Generacja klas anonimowych** (sekcja 3.1.7) — wykorzystanie `Symbol.newClass` do programatycznego tworzenia typów w czasie kompilacji
2. **Transformacja AST** (sekcja **??**) — przepisywanie właścicieli symboli (*re-owning*) poprzez `ReplaceRefs`
3. **Typy rafinowane** (sekcja 3.1.8) — dynamiczne rozszerzanie typów o pola strukturalne poprzez `Refinement`
4. **Walidacja 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:

- 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:

```
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 ten 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

Klasa replacerefs

Kluczową techniką jest zastąpienie referencji do starego kontekstu nowymi referencjami:

```
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 re-

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

Konstrukcja symbolu klasy

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)

Definicja członków klasy

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

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

Materializacja klasy

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:

- 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.

```scala
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 mapującej (np. `Map[String, Token]`) wymagałoby:

- Obliczenia funkcji haszującej dla klucza
- Przeszukiwania tablicy haszującej
- Potencjalnej obsługi kolizji
- Dynamicznego rzutowania typu

co wprowadzałoby 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:

- 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

Wady tego podejścia:

- 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

Wady tego podejścia:

- 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 raportuje 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:

```
    val root: Rule[Double] = rule { case Expr(e) => e }

    val Expr: Rule[Double] = rule(
      case (Expr(a), CalcLexer.PLUS(_), Term(b)) => a + b,
      case (Expr(a), CalcLexer.MINUS(_), Term(b)) => a - b,
```

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

Makro createTablesImpl

Centralnym elementem systemu jest makro `createTablesImpl`, które analizuje definicję parsera i generuje tabele w czasie kompilacji:

Makro wykonuje następujące kroki:

1. Ekstrakcja wszystkich reguł gramatycznych z definicji parsera poprzez refleksję TASTy
2. Transformacja wzorców dopasowania na produkcje gramatyczne
3. Konstrukcja automatów LR(1) i tabel parsowania
4. Generacja tabel akcji semantycznych
5. Walidacja gramatyki i rozwiązywanie konfliktów

Ekstrakcja produkcji z wzorców

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].

Rozwiązanie: 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:

- Tworzymy builder mapy jako zmienną lokalną (`Map.newBuilder`)
- Każde dodanie wpisu do buildera jest opakowane w osobną metodę `avoidTooLargeMethod()`
- Metody te są wywoływane sekwencyjnie jako lista wyrażeń w bloku
- 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:

- 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:

```
  def extractEBNF(ruleName: String)
    : PartialFunction[Expr[Rule[?]], Seq[(production: Production, action:
   Expr[Action[Ctx]])]] =
    case '{ rule(${ Varargs(cases) }*) } =>
      def createAction(binds: List[Option[Bind]], rhs: Term) =
   createLambda[Action[Ctx]]:
        case (methSym, (ctx: Term) :: (param: Term) :: Nil) =>
          val seqApplyMethod =
   param.select(TypeRepr.of[Seq[Any]].typeSymbol.methodMember("apply").head)
          val seq = param.asExprOf[Seq[Any]]

          val replacements = (find = ctxSymbol, replace = ctx) ::
            binds.zipWithIndex
              .collect:
                case (Some(bind), idx) => ((bind.symbol,
   bind.symbol.typeRef.asType), Expr(idx))
              .unsafeFlatMap:
```

Listing 3.15: Tworzenie akcji semantycznej z zachowaniem referencji

Kluczowe aspekty implementacji:

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
4. 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:

```
override val resolutions = Set(P.ofName("times").before(Lexer.PLUS),
   P.ofName("plus").after(Lexer.TIMES))
```

Listing 3.16: Deklaracja rozwiązań konfliktów

Implementacja wykorzystuje klasę `ConflictResolutionTable`, która podczas konstrukcji tabeli parsowania:

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

Implementacja ToExpr dla złożonych struktur

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:

- Tworzenie nowych symboli (`Symbol.newVal`) reprezentujących zmienne w generowanym kodzie
- Konstrukcja definicji wartości (`ValDef`) z przypisaniem początkowym
- Generacja listy wyrażeń manipulujących builderem

- 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.

Motywacja

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

- 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].

Definicja

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

Listing 3.18:]Definicja klasy typu Empty[T]

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.

Mechanizm derywacji

Derywacja instancji `Empty[T]` wykorzystuje mechanizm `Mirror.ProductOf[T]` 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.

Przykład użycia

```
case class Config(
  name: String = "default",
  count: Int = 0,
)
// Kompilator automatycznie derywuje instancje Empty[Config]
val empty = summon[Empty[Config]]
val instance: Config = empty() // Config("default", 0)
```

Listing 3.19:]Użycie Empty[T]

Alternatywy i ich ograniczenia

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.

Mechanizm `Empty[T]` eliminuje te problemy poprzez derywację w czasie kompilacji, zachowując bezpieczeństwo typów oraz zerowy narzut wykonania.

Ograniczenia

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.

Motywacja

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.

Definicja

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

Listing 3.20: Definicja klasy ReplaceRefs rozszerzającej TreeMap

Mechanizm działania

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.

Przykład użycia

Poniższy fragment ilustruje zastąpienie referencji do parametru makra `oldCtx` nowym symbolem `newCtx` w ciele wygenerowanej metody:

```
// Podczas ekspansji makra:
val oldCtxSymbol: Symbol = ... // Symbol parametru makra
val newCtxRef: Term = Ref(newCtxSymbol) // Referencja do nowego symbolu

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

// Transformacja ł ciaa funkcji:
val originalBody: Term = ... // ł Ciao funkcji ą odnoszce ę si do oldCtx
val transformedBody: Term = treeMap.transformTree(originalBody)(owner)
// Wszystkie ą wystpienia oldCtxSymbol ą s teraz ą zastpione 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].

Motywacja

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.

Definicja

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

Listing 3.22: Definicja klasy CreateLambda do programatycznej konstrukcji wyrażeń lambda

Mechanizm działania

Klasa `CreateLambda` implementuje algorytm konstrukcji wyrażeń lambda poprzez:

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.

Przykład użycia

```
1 val createLambda = CreateLambda()
2 val lambdaExpr: Expr[Int => Int] = createLambda[Int => Int] { case
      (methodSymbol, List(argTree)) =>
3 // Konstrukcja ł ciaa funkcji na podstawie symbolu metody i ó argumentw
      buildBody(argTree)
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.

Motywacja

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.

Definicja

```
@implicitNotFound("${T} should be a case class.")
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.

Mechanizm derywacji

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].

Przykład użycia

```
case class User(
  name: String,
  age: Int,
) // Kompilator automatycznie derywuje ęinstancj 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]

Uwaga techniczna

Operacja kopiowania realizowana przez `Copyable[T]` jest kopią płytką (*shallow copy*) — pola będące referencjami do obiektów wskazują na te same instancje w ory-

ginalnej 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, ogłaszany 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 w systemie ALPACA

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. Syntetyzacja: Decyzja projektu ALPACA

Powyższe rozważania 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 w systemie ALPACA

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 w ALPACA

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:

- Subsumpcję — sytuację, w której każde słowo akceptowane przez późniejszy wzorzec jest również akceptowane przez wcześniejszy wzorzec, czyniąc późniejszy „martwym kodem”. Przykład: jeśli definiuje się `ID = [a-z]+`, a następnie `KEYWORD = if|then`, to wszystkie słowa kluczowe będą dopasowane przez `ID`, co uniemożliwi rozpoznanie `KEYWORD`.

- Pokrycie prefiksów — sytuację, w której jeden wzorzec akceptuje prefiks słowa akceptowanego przez inny wzorzec. Przykład: jeśli zdefiniowano `LT = <` oraz `LE = <=`, lekser może dopasować `<`, a następnie zgłosić błąd leksykalny dla pozostałego znaku `=`, zamiast rozpoznać `<=`. Jest to konsekwencja użycia silnika wyrażeń regularnych, w którym zasada najdłuższego dopasowania (*maximal munch*) nie ma zastosowania.

Wykrycie któregokolwiek z powyższych problemów powoduje przerwanie kompilacji z czytelnym i działającym komunikatem diagnostycznym, dzięki czemu konfiguracyjne błędy są eliminowane przed czasem wykonania programu.

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*) [55] oraz na algorytmach predykcyjnych [56] lub analizie przesuwająco-redukcyjnej [57].

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*) [58]—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*) [59]—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 ze 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 [60]. Tabela akcji parsera LR mapuje parę (stan, prefiks wejścia) na akcję: *shift*, *re-*

duce, *accept*, lub *error*. Tabela `goto` określa przejścia po redukcjach do nowych stanów. W parserach LL analogiczną rolę spełnia tabela predykcji (nieterminal × prefiks wejścia → produkcja) [58].

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 [61]. 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. [translate:`if--then`] vs. [translate:`if--then--else`]) oraz z niejednoznaczności wyrażeń arytmetycznych [58].

5.2. Dobór klasy parsera w systemie ALPACA

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.

5.2.1. Zalety podejścia LR(1) w ALPACA

- 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.

5.2.2. Koszty i ograniczenia

- 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 [62].
- Parser jest deterministyczny na poziomie składniowym, lecz wciąż muszą zostać rozwiązane niejednoznaczności (np. [translate:`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) w ALPACA

Generowanie parsera LR(1) w ALPACA stanowi sekwencję algorytmów realizowanych w czasie kompilacji. Poniżej 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) [63]. `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).

```
1  @tailrec
2  private def addImports(firstSet: FirstSet, production: Production):
    FirstSet = production match {
3    case Production.NonEmpty(lhs, NonEmptyList(head: Terminal, tail)) =>
4      firstSet.updated(lhs, firstSet(lhs) + head)
5
6    case Production.NonEmpty(lhs, NonEmptyList(head: NonTerminal, tail)) =>
7      val newFirstSet = firstSet.updated(lhs, firstSet(lhs) ++
   (firstSet(head) - Symbol.Empty))
8
9      val nextProduction = tail match
10       case head :: next => Production.NonEmpty(lhs, NonEmptyList(head,
   next*))
11       case Nil => Production.Empty(lhs)
12
13     if firstSet(head).contains(Symbol.Empty)
14     then addImports(newFirstSet, nextProduction)
15     else newFirstSet
16
```

```
    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* [64] 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:

- Oblicza zbiór możliwych prefiksów wejścia: $FIRST(\beta\, a) = FIRST(\beta)$ bez ε, a jeśli $\varepsilon \in FIRST(\beta)$, dodaje także a.
- Dla każdego prefiksu wejścia x dodaje elementy $B \to \bullet\, \gamma\ ,\ x$ dla wszystkich produkcji $B \to \gamma$.
- 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:

- Domknięcie elementu `S' → • root, $` tworzy stan 0 (stan początkowy automatu).
- 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.
- Elementy z kropką na końcu produkcji (produkcja całkowicie rozpoznana) dodają akcje *reduce* dla swoich prefiksów wejścia. Specjalny element `S' → root •, $` generuje akcję *accept* dla symbolu `$`.
- 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
  )

```

```
12 val states = mutable.ListBuffer(initialState)
13 val table = mutable.Map.empty[(state: Int, stepSymbol: Symbol),
       ParseAction]
14
15 while states.sizeIs > currStateId do
16   val currState = states(currStateId)
17
18   // redukcje i akceptacja
19   for item <- currState if item.isLastItem do
20       addToTable(item.lookAhead, Reduction(item.production))
21
22   // śprzejcia shift (goto)
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.

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

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

[57] 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.

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

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

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

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

[62] 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).

[63] 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.

[64] *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).

Spis tabel

Spis listingów