



Akademia Górnictwo-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

<b>1 Cel prac i wizja projektu</b>	<b>5</b>
1.1 Charakterystyka problemu . . . . .	5
1.2 Motywacja projektu . . . . .	5
1.3 Przegląd istniejących rozwiązań . . . . .	6
1.3.1 Lex, Yacc . . . . .	6
1.3.2 PLY, SLY . . . . .	7
1.3.3 ANTLR . . . . .	8
1.3.4 Scala parser combinators . . . . .	9
1.3.5 ScalaBison . . . . .	9
1.3.6 parboiled2 . . . . .	9
1.3.7 FastParse . . . . .	10
1.3.8 Podsumowanie . . . . .	10
<b>2 Metaprogramowanie w Scali 3</b>	<b>12</b>
2.1 Wprowadzenie . . . . .	12
2.1.1 Quotes i Splices . . . . .	12
2.1.2 Bezpieczeństwo międzyetapowe . . . . .	12
2.2 Mechanizmy metaprogramowania w Scali 3 . . . . .	13
2.2.1 Definicje inline . . . . .	13
2.2.2 Makra oparte na wyrażeniach . . . . .	13
2.2.3 Dopasowanie wzorców kodu . . . . .	13
2.2.4 Refleksja TASTy . . . . .	13
<b>3 Implementacja</b>	<b>14</b>
3.1 Praktyczna implementacja analizatora leksykalnego z wykorzystaniem makr w Scali 3 . . . . .	14
3.1.1 Wprowadzenie do studium przypadku . . . . .	14
3.1.2 Architektura systemu leksera . . . . .	14
3.1.3 Analiza drzewa składni abstrakcyjnej . . . . .	15
3.1.4 Transformacja i adaptacja referencji . . . . .	15
3.1.5 Ekstrakcja i kompilacja wzorców . . . . .	16
3.1.6 Analiza wzorców: klasa CompileNameAndPattern . . . . .	16
3.1.7 Generacja klasy anonimowej . . . . .	17
3.1.8 Walidacja i obsługa błędów . . . . .	19
3.2 Praktyczna implementacja generatora parserów z wykorzystaniem makr w Scali 3 . . . . .	19
3.2.1 Wprowadzenie do generatora parserów . . . . .	19
3.2.2 Interfejs API parsera . . . . .	19

3.2.3	Generacja tabel parsowania w czasie kompilacji . . . . .	20
3.2.4	Trudne problemy rozwiążane w implementacji . . . . .	21
3.2.5	Generacja kodu tabel . . . . .	24
3.2.6	Podsumowanie implementacji parsera . . . . .	25
<b>Spis rysunków</b>		<b>28</b>
<b>Spis tabel</b>		<b>29</b>
<b>Spis algorytmów</b>		<b>30</b>
<b>Spis listingów</b>		<b>31</b>

# Rozdział 1

## Cel prac i wizja projektu

### 1.1. Charakterystyka problemu

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

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

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

### 1.2. Motywacja projektu

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

1. Stworzenie intuicyjnego API.
2. Opracowanie obszernej dokumentacji.
3. Rozbudowana diagnostyka błędów.
4. Poprawa wydajności względem rozwiązań w języku Python.
5. Integracja z popularnymi środowiskami programistycznymi (IDE).

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

## 1.3. Przegląd istniejących rozwiązań

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

### 1.3.1. Lex, Yacc

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

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

```

1 {
2 /*%%*/
3 value_expr($3);
4 $1->nd_value = $3;
5 $$ = $1;
6 /*
7 $$ = dispatch2(massign, $1, $3);
8 */
9 }
10 | var_lhs tOP_ASSIGN command_call
11 {
12 value_expr($3);
13 $$ = new_op_assign($1, $2, $3);
14 }
15 | primary_value '[' opt_call_args rbracket tOP_ASSIGN command_call
16 {
17 /*%%*/
18 NODE *args;
19
20 value_expr($6);
21 if (!$3) $3 = NEW_ZARRAY();
22 args = arg_concat($3, $6);
23 if ($5 == tROP) {
24     $5 = 0;
25 }
26 else if ($5 == tANDOP) {
27     $5 = 1;
28 }
29 $$ = NEW_OP_ASSIGN($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 w technologii Yacc

### 1.3.2. PLY, SLY

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

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

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

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

```

1 class MatrixParser(Parser):
2     tokens = MatrixScanner.tokens
3
4     precedence = (
5         ('nonassoc', 'IFX'),
6         ('nonassoc', 'ELSE'),
7         ('nonassoc', 'EQUAL'),
8     )
9
10    @_('{ " instructions " }')
11    def block(self, p: YaccProduction):
12        raise NotImplementedError
13
14    @_('instruction')
15    def block(self, p: YaccProduction):
16        raise NotImplementedError
17
18    @_('IF "(" condition ")" block %prec IFX')
19    def instruction(self, p: YaccProduction):

```

```

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

```

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

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

```

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

```

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

```

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

```

Listing 1.4: Przykładowy komunikat błędu w bibliotece *SLY*

### 1.3.3. ANTLR

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

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

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

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

### 1.3.4. Scala parser combinators

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

### 1.3.5. ScalaBison

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

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

### 1.3.6. parboiled2

*parboiled2*<sup>[10]</sup> 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 komplikacji i tłumaczy definicję reguły gramatycznej na odpowiadający jej bytecode JVM. Niestety próg wejścia ze względu na skomplikowany i nieintuicyjny DSL jest wysoki. Zgodnie z przykładem 1.5, raportowanie błędów jest bardzo ograniczone (problem z implementacją wynika jedynie z różnic w liczbie parametrów funkcji).

```

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

```

```

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

```

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

### 1.3.7. FastParse

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

### 1.3.8. Podsumowanie

Narzędzie	<b>Lex&amp;Yacc</b>	<b>PLY/SLY</b>	<b>ANTLR</b>	<b>scala-bison</b>
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	<b>Scala parser combinators</b>	<b>parboiled2</b>	<b>FastParse</b>	<b>ALPACA</b>
Język implementacji	Scala	Scala	Scala	Scala
Język użycia	DSL w Scali	DSL w Scali	DSL w Scali	Scala
Wydajność	wysoka	umiarkowana	wysoka	TODO
Łatwość użycia	niska	średnia	średnia	TODO
Aktywne wsparcie	nie	nie	tak	TODO
Diagnostyka błędów	dobra	niska	dobra	TODO
Dokumentacja	słaba	bardzo dobra	bardzo dobra	TODO
Popularność	średnia	niska	rosnąca	TODO
Integracja IDE	wsparcie dla Scali	wsparcie dla Scali	wsparcie dla Scali	TODO
Wsparcie do debugowania	dobre	dobre	dobre	TODO
Generowanie kodu	nie	nie	nie	TODO

Tabela 1.1: Porównanie wybranych narzędzi do generowania lekserów i parserów

# 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[14, 15].

Metaprogramowanie w Scali 3 zostało zaprojektowane z naciskiem na bezpieczeństwo typów, przenośność oraz skalowalność, oferując programistom możliwość generowania i analizowania kodu w czasie komplikacji przy zachowaniu pełnej ekspresywności języka[16, 17]. W przeciwieństwie do poprzedniego systemu, który eksponował wewnętrzne mechanizmy kompilatora i był źródłem problemów z kompatybilnością między wersjami[18], 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*), paradymat pozwalający na odróżnienie różnych etapów wykonania programu[19, 18]. W tym modelu kod może być wykonywany w różnych fazach: w czasie komplikacji (ang. *compile-time*) lub w czasie wykonania (ang. *runtime*)[19].

#### 2.1.1. Quotes i Splices

Kluczowymi koncepcjami w systemie metaprogramowania Scali 3 są *quotes* i *splices*[20, 21]. *Quotes*, oznaczane jako '`{ ... }`', służą do opóźnienia wykonania kodu i traktowania go jako danych[22, 23]. *Splices*, oznaczane jako '`$( ... )`', pozwalają na ocenę wyrażenia generującego kod i wstawienie wyniku do otaczającego kontekstu[22, 23, 24].

Formalna semantyka tych konstrukcji została przedstawiona w pracy Stuckiego, Brachthäusera i Odersky'ego[21], gdzie *quotes* i *splices* są traktowane jako prymitywne formy w typowanych drzewach składniowych (ang. *typed abstract syntax trees*). Autorzy dowodzą, że system zachowuje bezpieczeństwo typów oraz higienicznosć, zapewniając, że wygenerowany kod nie może przypadkowo powiązać identyfikatorów z niewłaściwymi zmiennymi[21].

#### 2.1.2. Bezpieczeństwo międzyetapowe

Scala 3 gwarantuje bezpieczeństwo międzyetapowe (ang. *cross-stage safety*) poprzez sprawdzanie poziomów etapowania w czasie komplikacji[18, 21]. 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[18].

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*)[18]. To podejście rozwiązuje problem wymazywania typów generycznych w JVM, zachowując informację o typach potrzebną w kolejnych etapach komplikacji.

## 2.2. Mechanizmy metaprogramowania w Scali 3

### 2.2.1. Definicje inline

Najprostszym narzędziem metaprogramowania jest modyfikator **inline**. Gwarantuje on, że wywołanie oznaczonej nim metody lub wartości zostanie w całości **wstawione w miejscu wywołania** (ang. *Inlining*) podczas komplikacji. Jest to polecenie dla kompilatora, a nie tylko sugestia, jak w niektórych innych językach.

### 2.2.2. Makra oparte na wyrażeniach

Makra w Scali 3 są zdefiniowane jako metody **inline** zawierające *splice* najwyższego poziomu (ang. *top-level splice*)[25, 26], czyli taki, który nie jest zagnieżdżony w żadnym *Quotes* i jest wykonywany w czasie komplikacji[19, 25].

Typ **Expr[T]** reprezentuje wyrażenie Scali o typie T jako typowane drzewo składowe[22, 26]. Makra manipulują wartościami typu **Expr[T]**, transformując je lub generując nowe wyrażenia[26]. Ta reprezentacja gwarantuje bezpieczeństwo typów na poziomie języka metaprogramowania[22].

### 2.2.3. Dopasowanie wzorców kodu

Scala 3 wspiera analizę kodu poprzez dopasowanie wzorców w *quotes* (ang. *quote pattern matching*)[18, 21]. Mechanizm ten pozwala na dekonstrukcję kawałków kodu i ekstrakcję podwyrażeń[21].

Stucki, Brachthäuser i Odersky[21] 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[21]. System gwarantuje, że ekstrahowane wyrażenia są zamknięte względem definicji wewnętrz wzorca, zapobiegając wyciekom zakresu[21].

### 2.2.4. Refleksja TASTy

Dla przypadków wymagających głębszej analizy kodu, Scala 3 oferuje API refleksji TASTy[22, 23, 27]. TASTy jest binarnym formatem serializacji typowanych drzew składowych używanym przez kompilator Scali 3[18].

API refleksji dostarcza szczegółowy widok na strukturę kodu, włączając typy, symbole oraz pozycje w kodzie źródłowym[22, 27]. Jest dostępne poprzez obiekt **reflect** zdefiniowany w typie **Quotes**, który jest przekazywany kontekstualnie do makr[22, 27].

# Rozdział 3

## Implementacja

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

#### 3.1.1. Wprowadzenie do studium przypadku

Niniejszy rozdział prezentuje praktyczną implementację systemu analizy leksykalnej (leksera) wykorzystującego zaawansowane mechanizmy metaprogramowania Scali 3. Przedstawiony kod stanowi przykład zastosowania technik opisanych w poprzednich rozdziałach do rozwiązania rzeczywistego problemu inżynierskiego: automatycznej generacji wydajnego analizatora leksykalnego z definicji wysokopoziomowej w formie języka dziedzinowego (DSL).

System **alpaca.lexer** implementuje transformację deklaratywnych reguł tokenizacji zapisanych jako funkcja częściowa (ang. *partial function*) w kod proceduralny wykonywany w czasie komplikacji. Wykorzystuje przy tym pełne spektrum możliwości refleksji TASTy, włączając generację klas w czasie komplikacji, transformację drzew AST oraz wypecializowane typy refinement.

#### 3.1.2. Architektura systemu leksera

##### Interfejs użytkownika

System oferuje użytkownikowi przejrzysty interfejs DSL oparty na dopasowaniu wzorców:

```
1 type LexerDefinition[Ctx <: AnyGlobalCtx] = 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.

Główny punkt wejścia systemu stanowi metoda **lexer**:

```
1 transparent inline def lexer[Ctx <: AnyGlobalCtx & Product](  
2   using Ctx WithDefault DefaultGlobalCtx,
```

```

3 | )(
4 |   inline rules: Ctx ?=> LexerDefinition[Ctx],
5 | )(using
6 |   copy: Copyable[Ctx],
7 |   betweenStages: BetweenStages[Ctx],
8 | )(using inline
9 |   debugSettings: DebugSettings[?, ?],

```

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.

### Implementacja makra

Makro przyjmuje wyrażenie reprezentujące reguły leksera jako **Expr[Ctx ?=> LexerDefinition[Ctx]]** oraz instancje kontekstualnych klas pomocniczych. Parametr **using Quotes** dostarcza dostępu do API refleksji TASTy.

### 3.1.3. Analiza drzewa składni abstrakcyjnej

#### Dekonstrukcja funkcji częściowej

Kluczowym krokiem implementacji jest ekstrakcja reguł z definicji funkcji częściowej:

```

1 | val createLambda = new CreateLambda[quotes.type]

```

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

Ten fragment kodu wykorzystuje dopasowanie wzorców w *quotes* do dekonstrukcji typowanego AST funkcji częściowej. Struktura **Lambda(\_, Match(\_, cases))** odpowiada wewnętrznej reprezentacji funkcji częściowej, gdzie **Match** zawiera listę przypadków **CaseDef**.

### 3.1.4. Transformacja i adaptacja referencji

#### Klasa replacerefs

Kluczową techniką jest zastąpienie referencji do starego kontekstu nowymi referencjami:

```

1 | val (tokens, infos) = cases.foldLeft((List.empty[Expr[ThisToken]], 
2 |   List.empty[TokenInfo[?]])):
3 |   case ((accTokens, accInfos), CaseDef(tree, None, body)) =>
4 |     def replaceWithNewCtx(newCtx: Term) = new
      ReplaceRefs[quotes.type].apply(
        find = oldCtx.symbol, replace = newCtx),

```

Listing 3.4: Zastąpienie referencji starego kontekstu nowymi (ReplaceRefs)

Transformacja ta realizuje proces znany jako "re-owning" w terminologii kompilatorów — zmianę właściciela (owner) symboli w AST. Jest to konieczne, ponieważ kod oryginalnie

odnoszący się do parametru makra musi zostać przepisany, aby odnosił się do parametru metody w wygenerowanej klasie.

Klasa **ReplaceRefs** udostępnia **TreeMap**, który podczas przejścia po AST podmienia referencje do wskazanych symboli na podane termy. Umożliwia to tzw. re-owning — przeniesienie fragmentów kodu między różnymi właścicielami symboli bez ręcznego przepisywania drzew (por. ??).

### 3.1.5. Ekstrakcja i komplikacja wzorców

#### Funkcja extractSimple

Funkcja **extractSimple** implementuje logikę dopasowania różnych typów definicji tokenów:

```

1      )
2
3  def extractSimple(
4    ctxManipulation: Expr[CtxManipulation[Ctx]],
5  ): PartialFunction[Expr[ThisToken], List[Expr[ThisToken]]] =
6    case '{ Token.Ignored(using $ctx) } =>
7
8      case '{ type t <: ValidName; Token.apply[t]($value: v)(using $ctx)
9    } =>
10       compileNameAndPattern[t](tree).map:
11         case '{ $tokenInfo: TokenInfo[name] } =>
12           // we need to widen here to avoid weird types
13           TypeRepr.of[v].widen.asType match
14             case '[result] =>
15               val remapping = createLambda[Ctx => result]:
16                 case (methSym, (newCtx: Term) :: Nil) =>
17
18               replaceWithNewCtx(newCtx).transformTerm(value.asTerm)(methSym)

```

Listing 3.5: Funkcja extractSimple: dopasowywanie definicji tokenów

Wykorzystuje ona dopasowanie wzorców w *quotes* z ekstraktorem typów, 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.

### 3.1.6. Analiza wzorców: klasa **CompileNameAndPattern**

Klasa **CompileNameAndPattern** stanowi kluczowy komponent systemu analizy lekkośkalnej, odpowiedzialny za ekstrakcję i walidację wzorców tokenów podczas ekspansji makra. Jej głównym zadaniem jest transformacja różnorodnych form wzorców występujących w definicjach DSL na ujednolicone struktury **TokenInfo**, które następnie są wykorzystywane do generacji finalnego kodu leksera.

Implementacja wykorzystuje rekurencyjne przetwarzanie drzewa AST z zastosowaniem optymalizacji rekurencji ogonowej (**@tailrec**), co zapewnia efektywność działania nawet dla złożonych wzorców z wieloma alternatywami.

### 3.1.7. Generacja klasy anonimowej

Anonimowa klasa implementująca `Tokenization[Ctx]` jest konstruowana programatycznie: (1) tworzymy symbol klasy przez `Symbol.newClass` wraz z listą deklaracji pól i typów; (2) budujemy ciało klasy (`ClassDef`) zawierające `ValDef` dla każdego zdefiniowanego tokena oraz pola `tokens` i `byName`; (3) określamy rodzica przez wywołanie konstruktora `Tokenization[Ctx]` z wymaganymi zależnościami; (4) instancjonujemy klasę i nadajemy jej typ zrafinowany przez kolejne `Refinement` odpowiadające polom-tokenom.

```

1      name = "byName",
2      tpe = TypeRepr.of[Map[String, DefinedToken[?, Ctx, ?]]],
3      flags = Flags.Synthetic | Flags.Lazy, // todo: reconsider lazy
4      privateWithin = Symbol.noSymbol,
5    )
6
7    tokenDecls ++ List(fieldsDecls, compiled, allTokens, byName)
8  }
9
10 val cls = Symbol.newClass(
11   Symbol.spliceOwner,
12   Symbol.freshName("$anon"),
13   List(TypeRepr.of[Tokenization[Ctx]]),
14   decls,
15   None,
16 )
17
18 val body = {
19   val tokenVals = definedTokens.collect:
20     case '{ $token: DefinedToken[name, Ctx, value] } =>
21       ValDef(
22         cls.fieldMember(ValidName.typeToString[name]),
23         Some(token.asTerm.changeOwner(cls.fieldMember(ValidName.typeToString[name]))),
24       )
25
26   tokenVals ++ Vector(
27     TypeDef(cls.typeMember("Fields")),
28     ValDef(
29       cls.fieldMember("compiled"),
30       Some {
31         val regex = Expr(
32           infos
33             .map:
34               case TokenInfo(_, regexGroupName, pattern) =>
35               s"(?<$regexGroupName>$pattern)"
36               .mkString("|")
37               .r
38               .regex, // we'd like to compile it here to fail in compile
39               time if regex is invalid
40             )
41         '{ Regex($regex)
42       }.asTerm.changeOwner(cls.fieldMember("compiled"))
43       },
44     ),
45     ValDef(

```

```

44     cls.fieldMember("tokens"),
45     Some {
46       val declaredTokens = definedTokens.map:
47         case '{ $token: DefinedToken[name, Ctx, ?] } =>
48
49       This(cls).select(cls.fieldMember(ValidName.typeToString[name])).asExprOf[ThisToken]
50
51       Expr.ofList(ignoredTokens ++
52       declaredTokens).asTerm.changeOwner(cls.fieldMember("tokens"))
53     },
54   ValDef(
55     cls.fieldMember("byName"),
56     Some {
57       val all = Expr.ofSeq {
58         tokenVals.map(valDef => Expr.ofTuple((Expr(valDef.name),
59         Ref(valDef.symbol).asExprOf[DefinedToken[?, Ctx, ?]])))
60       }
61
62       '{ Map($all*) }.asTerm
63     },
64   )
65 }
66 val tokenizationConstructor =
67   TypeRepr.of[Tokenization[Ctx]].typeSymbol.primaryConstructor
68
69 val parents =
70   New(TypeTree.of[Tokenization[Ctx]])
71     .select(tokenizationConstructor)
72     .appliedToType(TypeRepr.of[Ctx])
73     .appliedToArgs(List(copy.asTerm, betweenStages.asTerm)) :: Nil
74

```

Listing 3.6: Generacja i instancjowanie anonimowej klasy `Tokenization[Ctx]` z typem rafinowanym

## Typy refinement

Zwracany typ jest stopniowo rafinowany dla każdego tokena:

```

1   ),
2   )
3 }
4
5 val tokenizationConstructor =
6   TypeRepr.of[Tokenization[Ctx]].typeSymbol.primaryConstructor

```

Listing 3.7: Rafinowanie typu wynikowego o pola tokenów

`Refinement(tpe, name, memberType)` tworzy typ refinement dodający członka o podanej nazwie i typie do bazowego typu. Pozwala to kompilatorowi śledzić, że zwrócony obiekt ma pola odpowiadające poszczególnym tokenom, umożliwiając dostęp do nich z pełnym wsparciem systemu typów.

### 3.1.8. Walidacja i obsługa błędów

#### Walidacja wzorców regularnych

System wykorzystuje pomocniczą klasę **RegexChecker** do walidacji wzorców:

Poniższy mechanizm sprawdza poprawność składni wyrażeń regularnych już w czasie komplikacji i raportuje błędy z dokładną lokalizacją wzorca. Metoda **report.errorAndAbort** jest częścią API kompilatora do raportowania błędów w czasie komplikacji. Przerwanie komplikacji w przypadku niepoprawnych wzorców zapewnia, że błędy konfiguracji są wykrywane możliwie wcześnie.

#### Obsługa nieobsługiwanych konstrukcji

Kod jawnie sygnalizuje nieobsługiwane przypadki: Obsługiwane są wyłącznie jasno zdefiniowane formy wzorców; w przypadku napotkania innej konstrukcji komplikacja 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

System **alpaca.parser** stanowi zaawansowaną implementację generatora parserów LR(1), wykorzystującego metaprogramowanie Scali 3 do automatycznej konstrukcji tabel parsowania w czasie komplikacji. W przeciwieństwie do tradycyjnych generatorów parserów takich jak Yacc[2], które generują kod w osobnym kroku komplikacji, system ALPACA integruje generację parsera bezpośrednio w procesie komplikacji Scali, oferując pełne wsparcie systemu typów oraz natychmiastową walidację gramatyk.

Implementacja parsera reprezentuje znacznie bardziej złożone wyzwanie inżynierskie niż lekser, ze względu na konieczność: (1) konstrukcji automatów LR(1) i tabel parsowania, (2) obsługi konfliktów shift-reduce i reduce-reduce, (3) generacji akcji semantycznych zachowujących bezpieczeństwo typów, (4) oraz radzenia sobie z ograniczeniami platformy JVM dotyczącymi rozmiaru metod.

### 3.2.2. Interfejs API parsera

#### Definicja parsera

Użytkownik definiuje parser poprzez dziedziczenie po klasie bazowej **Parser[Ctx]**:

```

1 abstract class Parser[Ctx <: AnyGlobalCtx]{
2   using Ctx WithDefault EmptyGlobalCtx,
3 }(using
4   empty: Empty[Ctx],
5   tables: Tables[Ctx],
6 ) {

```

Listing 3.8: 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:

```

1 object CalcParser extends Parser[EmptyGlobalCtx] {
2   val Expr: Rule[Int] = rule(
3     { case (Expr(a), Lexer.PLUS(_), Expr(b)) => a + b },
4     { case Lexer.NUMBER(n) => n.value }
5   )
6
7   val root = rule { case Expr(result) => result }
8 }
```

Listing 3.9: 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), Lexer.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:

```

1 private def createTablesImpl[Ctx <: AnyGlobalCtx: Type](
2   using quotes: Quotes,
3 )(implicit
4   debugSettings: Expr[DebugSettings[?, ?]],
5 ): Expr[(parseTable: ParseTable, actionTable: ActionTable[Ctx])] = {
```

Listing 3.10: Sygnatura makra `createTablesImpl`

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:

```

1 def extractEBNF(ruleName: String): Expr[Rule[?]] => Seq[(production:
2   Production, action: Expr[Action[Ctx]])] = {
3   case '{ rule(${ Varargs(cases) }*) } =>
4     def createAction(binds: List[Option[Bind]], rhs: Term) =
5       createLambda[Action[Ctx]]:
6         case (methSym, (ctx: Term) :: (param: Term) :: Nil) =>
7           val seqApplyMethod =
8             param.select(TypeRepr.of[Seq[Any]].typeSymbol.methodMember("apply").head)
9             val seq = param.asExprOf[Seq[Any]]
10
11           val replacements = (find = ctxSymbol, replace = ctx) ::=
12             binds.zipWithIndex
13             .collect:
14               case (Some(bind), idx) => ((bind.symbol,
15                 bind.symbol.typeRef.asType), Expr(idx))
16               .map:
17                 case ((bind, '[t]), idx) => (find = bind, replace = ${
18                   $seq($idx).asInstanceOf[t].asTerm)
19                 case x => raiseShouldNeverBeCalled(x.toString)
20
21             replaceRefs(replacements*).transformTerm(rhs)(methSym)
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
168
169
169
170
171
172
173
174
175
176
177
178
179
179
180
181
182
183
184
185
186
187
187
188
189
189
190
191
192
193
194
195
196
197
197
198
199
199
200
201
202
203
203
204
205
205
206
206
207
207
208
208
209
209
210
210
211
211
212
212
213
213
214
214
215
215
216
216
217
217
218
218
219
219
220
220
221
221
222
222
223
223
224
224
225
225
226
226
227
227
228
228
229
229
230
230
231
231
232
232
233
233
234
234
235
235
236
236
237
237
238
238
239
239
240
240
241
241
242
242
243
243
244
244
245
245
246
246
247
247
248
248
249
249
250
250
251
251
252
252
253
253
254
254
255
255
256
256
257
257
258
258
259
259
260
260
261
261
262
262
263
263
264
264
265
265
266
266
267
267
268
268
269
269
270
270
271
271
272
272
273
273
274
274
275
275
276
276
277
277
278
278
279
279
280
280
281
281
282
282
283
283
284
284
285
285
286
286
287
287
288
288
289
289
290
290
291
291
292
292
293
293
294
294
295
295
296
296
297
297
298
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
13
```

Takie podejście generuje pojedynczą, dużą metodę zawierającą wszystkie wpisy tabeli, co dla gramatyk o rozmiarze produkcyjnym skutkuje błędem komplikacji **java.lang.ClassFormatError: Code length too large**.

**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:

```

1  val additions = entries
2    .map(entry =>
3      '{'
4        def avoidTooLargerMethod(): Unit = $builder += ${ Expr(entry) }
5        avoidTooLargerMethod()
6        }.asTerm,
7      )
8    .toList

```

Listing 3.13: 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 **avoidTooLargerMethod()**
- Metody te są wywoływanie sekwencyjnie jako lista wyrażeń w bloku
- Końcowy wynik jest uzyskiwany przez wywołanie **builder.result()**

Ta technika skutecznie obchodzi limit rozmiaru metody, ponieważ każda pomocnicza metoda zawiera tylko kilka instrukcji bajtowych, niezależnie od rozmiaru całej tabeli. Dodatkowo, kompilator JIT może efektywnie zoptymalizować i zliniować te małe metody w czasie wykonania, eliminując narzut wydajnościowy.

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:

```

1     def createAction(binds: List[Option[Bind]], rhs: Term) =
2       createLambda[Action[Ctx]]:
3         case (methSym, (ctx: Term) :: (param: Term) :: Nil) =>
4           val seqApplyMethod =
5             param.select(TypeRepr.of[Seq[Any]].typeSymbol.methodMember("apply").head)
6             val seq = param.asExprOf[Seq[Any]]
7
7           val replacements = (find = ctxSymbol, replace = ctx) ::=
8             binds.zipWithIndex
9               .collect:
10                 case (Some(bind), idx) => ((bind.symbol,
11                   bind.symbol.typeRef.asType), Expr(idx))
12                   .map:
13                     case ((bind, '[t]), idx) => (find = bind, replace = '{ $seq($idx).asInstanceOf[t] }.asTerm)
14                     case x => raiseShouldNeverBeCalled(x.toString)
15
16           replaceRefs(replacements*).transformTerm(rhs)(methSym)

```

Listing 3.14: Tworzenie akcji semantycznej z zachowaniem referencji

Kluczowe aspekty implementacji:

- Parametryzacja akcji:** Akcja jest transformowana w funkcję przyjmującą kontekst (**ctx**) oraz listę dzieci w drzewie parsowania (**param**)
- Re-owning symboli:** Referencje do kontekstu parsera są zastępowane parametrem funkcji
- Ekstrakcja wartości:** Wartości z dopasowanych symboli są ekstrahowane z listy dzieci poprzez indeksowanie
- Rzutowanie typów:** System typów zapewnia, że ekstrakcje są bezpieczne względem typów dzięki informacji z wzorca dopasowania

## Rozwiązywanie konfliktów gramatycznych

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

```

1 override val resolutions = Set(
2   P.ofName("times").before(Lexer.PLUS),
3   P.ofName("plus").after(Lexer.TIMES)
4 )

```

Listing 3.15: Deklaracja rozwiązań konfliktów

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

- Wykrywa konflikty między akcjami dla danego stanu i symbolu
- Konsultuje zdefiniowane przez użytkownika relacje precedencji
- Wybiera odpowiednią akcję zgodnie z deklaracją

4. Zgłasza błąd komplikacji dla nieroziązanych konfliktów

To podejście umożliwia wyrażenie precedencji i łączności operatorów w sposób bardziej naturalny niż tradycyjne deklaracje `%left`, `%right` i `%nonassoc` w Yacc.

### 3.2.5. Generacja kodu tabel

#### Implementacja `ToExpr` dla złożonych struktur

System wymaga konwersji struktur danych w czasie komplikacji (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):

```

1  given ToExpr[ParseTable] with
2    def apply(entries: ParseTable)(using quotes: Quotes): Expr[ParseTable]
3    = {
4      import quotes.reflect./*
5
6      type BuilderTpe = mutable.Builder[
7        ((state: Int, stepSymbol: alpaca.parser.Symbol), Shift | Reduction),
8        Map[(state: Int, stepSymbol: alpaca.parser.Symbol), Shift | Reduction],
9        ]
10
11     val symbol = Symbol.newVal(
12       Symbol.spliceOwner,
13       Symbol.freshName("builder"),
14       TypeRepr.of[BuilderTpe],
15       Flags.Mutable,
16       Symbol.noSymbol,
17     )
18
19     val valDef = ValDef(symbol, Some('{ Map.newBuilder: BuilderTpe
20   }.asTerm))
21
22     val builder = Ref(symbol).asExprOf[BuilderTpe]
23
24     val additions = entries
25     .map(entry =>
26       '{{
27         def avoidTooLargerMethod(): Unit = $builder += ${Expr(entry)}
28         avoidTooLargerMethod()
29       }.asTerm,
30     )
31     .toList
32
33     val result = '{ $builder.result() }.asTerm
34
35     Block(valDef :: additions, result).asExprOf[ParseTable]
36   }

```

Listing 3.16: 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

### 3.2.6. Podsumowanie implementacji parsera

Implementacja generatora parserów w systemie ALPACA demonstruje zaawansowane zastosowanie metaprogramowania Scali 3 do rozwiązywania rzeczywistych problemów inżynierskich. Kluczowe osiągnięcia obejmują:

- **Automatyczna konstrukcja tabel LR(1)** w czasie komplikacji eliminująca potrzebę osobnego kroku generacji kodu
- **Pełna integracja z systemem typów Scali**, zapewniająca bezpieczeństwo typów w akcjach semantycznych
- **Eleganckie obejście ograniczeń platformy JVM** poprzez inteligentną fragmentację generowanego kodu
- **Deklaratywny mechanizm rozwiązywania konfliktów** oferujący większą elastyczność niż tradycyjne narzędzia
- **Natychmiastowa walidacja gramatyk** podczas komplikacji, z czytelnymi komunikatami o błędach

Problemy napotkane i rozwiążane podczas implementacji, szczególnie ograniczenie rozmiaru metod JVM, ilustrują istotne aspekty praktycznego metaprogramowania: generowany kod musi nie tylko być poprawny funkcjonalnie, ale również respektować wszystkie techniczne ograniczenia platformy docelowej.

# Bibliografia

- [1] M. E. Lesk i E. Schmidt. *Lex: A lexical analyzer generator*. T. 39. Bell Laboratories Murray Hill, NJ, 1975.
- [2] S. C. Johnson i in. *Yacc: Yet another compiler-compiler*. T. 32. Bell Laboratories Murray Hill, NJ, 1975.
- [3] D. Beazley. *PLY (Python Lex-Yacc)*. 2005. URL: <https://www.dabeaz.com/ply/ply.html> (term. wiz. 19.03.2025).
- [4] D. Beazley. *SLY (Sly Lex-Yacc)*. 2016. URL: <https://sly.readthedocs.io/en/latest/sly.html> (term. wiz. 19.03.2025).
- [5] D. Beazley. *SLY Github*. URL: <https://github.com/dabeaz/sly> (term. wiz. 19.03.2025).
- [6] T. Parr, P. Wells, R. Klaren, L. Craymer, J. Coker, S. Stanchfield, J. Mitchell i C. Flack. *What's ANTLR*. 2004.
- [7] A. Moors, F. Piessens i M. Odersky. „Parser combinators in Scala”. W: *CW Reports* (2008).
- [8] *scala-parser-combinators Getting Started*. URL: [https://github.com/scala/scala-parser-combinators/blob/main/docs/Getting\\_Started.md](https://github.com/scala/scala-parser-combinators/blob/main/docs/Getting_Started.md) (term. wiz. 04.04.2025).
- [9] J. Boyland i D. Spiewak. „Tool paper: ScalaBison recursive ascent-descent parser generator”. W: *Electronic Notes in Theoretical Computer Science* 253.7 (2010).
- [10] A. A. Myltsev. „Parboiled2: A macro-based approach for effective generators of parsing expressions grammars in Scala”. W: *arXiv preprint arXiv:1907.03436* (2019).
- [11] L. Haoyi. *sfscala.org: Li Haoyi, FastParse: Fast, Programmable, Modern Parser-Combinators in Scala*. 2015.
- [12] *FastParse Getting Started*. URL: <https://com-lihaoyi.github.io/fastparse/#GettingStarted> (term. wiz. 04.04.2025).
- [13] L. Haoyi. *FastParse. Fast, Modern Parser Combinators*. URL: <https://www.lihaoyi.com/post/slides/FastParse.pdf> (term. wiz. 18.04.2025).
- [14] *Dropped: Scala 2 Macros. Scala 3 Reference*. URL: <https://docs.scala-lang.org/scala3/reference/metaprogramming/migrating/macros-compatibility.html> (term. wiz. 25.10.2025).
- [15] *Metaprogramming. Scala 3 Reference*. URL: <https://docs.scala-lang.org/scala3/reference/metaprogramming.html> (term. wiz. 25.10.2025).
- [16] 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).

- 
- [17] 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](https://doi.org/10.1145/3426426.3428486). URL: <https://dl.acm.org/doi/10.1145/3426426.3428486>.
  - [18] N. Stucki. „Scalable Metaprogramming in Scala 3”. Prac. dokt. Lausanne: EPFL, 2020.
  - [19] *Runtime Multi-Stage Programming. Scala 3 Reference*. URL: <https://docs.scala-lang.org/scala3/reference/metaprogramming/staging.html> (term. wiz. 25. 10. 2025).
  - [20] N. Stucki, J. Brachth”auser 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](https://doi.org/10.1145/3278122.3278139). URL: <https://dl.acm.org/doi/10.1145/3278122.3278139>.
  - [21] 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](https://doi.org/10.1145/3486609.3487203). URL: <https://dl.acm.org/doi/10.1145/3486609.3487203>.
  - [22] *Reflection. Scala 3 Reference*. URL: <https://docs.scala-lang.org/scala3/reference/metaprogramming/reflection.html> (term. wiz. 25. 10. 2025).
  - [23] *Reflection - Scala 3 - EPFL (Dotty docs)*. URL: <https://dotty.epfl.ch/docs/reference/metaprogramming/reflection.html> (term. wiz. 25. 10. 2025).
  - [24] *Quoted Code / Macros in Scala 3*. URL: <https://docs.scala-lang.org/scala3/guides/macros/quotes.html> (term. wiz. 25. 10. 2025).
  - [25] *Macros. Scala 3 Reference*. URL: <https://docs.scala-lang.org/scala3/reference/metaprogramming/macros.html> (term. wiz. 25. 10. 2025).
  - [26] *Scala 3 Macros*. URL: <https://docs.scala-lang.org/scala3/guides/macros/macros.html> (term. wiz. 25. 10. 2025).
  - [27] *Reflection / Macros in Scala 3*. URL: <https://docs.scala-lang.org/scala3/guides/macros/reflection.html> (term. wiz. 25. 10. 2025).
  - [28] 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.

# **Spis rysunków**

# Spis tabel

1.1 Porównanie wybranych narzędzi do generowania lekserów i parserów . . . . 11

# **Spis algorytmów**

# Spis listingów

1.1	Fragment definicji parsera Ruby w technologii Yacc . . . . .	6
1.2	Fragment definicji parsera w Pythonie, wykorzystujacy bibliotekę SLY . . . . .	7
1.3	Fragment niedzialajacego kodu w Pythonie, wykorzystujacy bibliotekę SLY	8
1.4	Przykładowy komunikat błędu w bibliotece <i>SLY</i> . . . . .	8
1.5	Fragment błędu wygenerowanego przez bibliotekę <i>parboiled2</i> . . . . .	9
3.1	Definicja typu <i>LexerDefinition</i> . . . . .	14
3.2	Punkt wejścia: transparent inline def lexer . . . . .	14
3.3	Dekonstrukcja funkcji częściowej (dopasowanie AST do <i>CaseDef</i> ) . . . . .	15
3.4	Zastąpienie referencji starego kontekstu nowymi (ReplaceRefs) . . . . .	15
3.5	Funkcja <i>extractSimple</i> : dopasowywanie definicji tokenów . . . . .	16
3.6	Generacja i instancjowanie anonimowej klasy <b>Tokenization[Ctx]</b> z typem rafinowanym . . . . .	17
3.7	Rafinowanie typu wynikowego o pola tokenów . . . . .	18
3.8	Klasa bazowa <i>Parser</i> . . . . .	19
3.9	Przykład definicji reguł parsera . . . . .	20
3.10	Sygnatura makra <i>createTablesImpl</i> . . . . .	20
3.11	Ekstrakcja produkcji gramatycznych z wzorców . . . . .	21
3.12	Naiwna implementacja prowadząca do przekroczenia limitu . . . . .	21
3.13	Rozwiązywanie problemu rozmiaru metod przez fragmentację . . . . .	22
3.14	Tworzenie akcji semantycznej z zachowaniem referencji . . . . .	23
3.15	Deklaracja rozwiązań konfliktów . . . . .	23
3.16	Implementacja <i>ToExpr</i> dla <i>ParseTable</i> . . . . .	24