



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

1	Cel prac i wizja projektu	5
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
2	Metaprogramowanie w Scali 3	12
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
3	Implementacja	14
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	16
3.1.8	Typy rafinowane (refinement types)	17
3.1.9	Uzasadnienie wybranego podejścia implementacyjnego	19
3.1.10	Analiza alternatywnych rozwiązań	20
3.1.11	Walidacja i obsługa błędów	21
	Spis rysunków	24

Spis tabel	25
Spis algorytmów	26
Spis listingów	27

Rozdział 1

Cel prac i wizja projektu

1.1. Charakterystyka problemu

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

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

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

1.2. Motywacja projektu

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

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

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

1.3. Przegląd istniejących rozwiązań

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

1.3.1. Lex, Yacc

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

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

```

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_ASGN command_call
11 {
12  value_expr($3);
13  $$ = new_op_assign($1, $2, $3);
14  }
15 | primary_value '[' opt_call_args rbracket tOP_ASGN command_call
16 {
17  /*%%*/
18  NODE *args;
19
20  value_expr($6);
21  if (!$3) $3 = NEW_ZARRAY();
22  args = arg_concat($3, $6);
23  if ($5 == tOROP) {
24    $5 = 0;
25  }
26  else if ($5 == tANDOP) {
27    $5 = 1;
28  }
29  $$ = NEW_OP_ASGN1($1, $5, args);
30  fixpos($$, $1);
31  /*%
32  $$ = dispatch2(aref_field, $1, escape_Qundef($3));
33  $$ = dispatch3(opassign, $$, $5, $6);
34  %*/
35  }

```

Listing 1.1: Fragment definicji parsera Ruby 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 ?? ilustruje kilka nieintuicyjnych, automatycznych mechanizmów obecnych w bibliotece *SLY*.

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

```

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, wykorzystujący bibliotekę *SLY*

Komunikaty błędów w bibliotece *SLY* są bardzo ograniczone, co obrazuje przykład 1.3, który po uruchomieniu informuje użytkownika błędem z fragmentu kodu 1.3. 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 niedziałającego kodu w Pythonie, wykorzystujący bibliotekę *SLY*

```

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

```

Listing 1.4: Przykładowy komunikat błędu w bibliotece *SLY*

1.3.3. ANTLR

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

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

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

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

1.3.4. Scala parser combinators

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

1.3.5. ScalaBison

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

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

1.3.6. parboiled2

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

```

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

```

```

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

```

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

1.3.7. FastParse

FastParseFastParse[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 alternatyw, przy jednoczesnym zachowaniu czytelności zbliżonej do formalnych definicji gramatyki. Według dokumentacji[12], parsery *Fastparse* zajmują 1/10 kodu w porównaniu do ręcznie napisanego parsera rekurencyjnego. W porównaniu do narzędzi generujących parsery, takich jak *ANTLR* lub *Lex* i *Yacc*, implementacja nie wymaga żadnego specjalnego kroku kompilacji lub generowania kodu. To sprawia, że rozpoczęcie pracy z *Fastparse* jest znacznie łatwiejsze niż w przypadku bardziej tradycyjnych narzędzi do generowania parserów. Przykładowo, parser wyrażeń arytmetycznych może być zwięźle napisany, aby obsługiwać zagnieżdżone nawiasy, pierwszeństwo operatorów i raportowanie błędów w mniej niż 20 liniach kodu[13]. Biblioteka kładzie również nacisk na debugowanie, generując szczegółowe komunikaty o błędach, które wskazują dokładną lokalizację i przyczynę niepowodzeń parsowania, takich jak niedopasowane nawiasy lub nieprawidłowe tokeny.

1.3.8. Podsumowanie

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	dobrze	częściowe	dobrze
Generowania kodu	nie	nie	tak	nie
Narzędzie	Scala parser combinators	parboiled2	FastParse	ALPACA
Język implementacji	Scala	Scala	Scala	Scala
Język użycia	DSL w Scali	DSL w Scali	DSL w Scali	Scala
Wydajność	wysoka	umiarkowana	wysoka	TODO
Łatwość użycia	niska	średnia	średnia	TODO
Aktywne wsparcie	nie	nie	tak	TODO
Diagnostyka błędów	dobra	niska	dobra	TODO
Dokumentacja	słaba	bardzo dobra	bardzo dobra	TODO
Popularność	średnia	niska	rosnąca	TODO
Integracja IDE	wsparcie dla Scali	wsparcie dla Scali	wsparcie dla Scali	TODO
Wsparcie do debugowania	dobrze	dobrze	dobrze	TODO
Generowania 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 kompilacji 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*), paradygmat pozwalający na odróżnienie różnych etapów wykonania programu[19, 18]. W tym modelu kod może być wykonywany w różnych fazach: w czasie kompilacji (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]. *Splices*, oznaczane jako `$ { ... }`, pozwalają na ocenę wyrażenia generującego kod i wstawienie wyniku do otaczającego kontekstu[22, 23].

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 higieniczność, 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 kompilacji[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 kompilacji.

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 kompilacji. 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*)[24, 25], czyli taki, który nie jest zagnieżdżony w żadnym *Quotes* i jest wykonywany w czasie kompilacji[19, 24].

Typ `Expr[T]` reprezentuje wyrażenie Scali o typie `T` jako typowane drzewo składniowe[22, 25]. Makra manipulują wartościami typu `Expr[T]`, transformując je lub generując nowe wyrażenia[25]. 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. System gwarantuje, że ekstrahowane wyrażenia są zamknięte względem definicji wewnątrz wzorca, zapobiegając wyciekowi zakresu.

2.2.4. Refleksja TASTy

Dla przypadków wymagających głębszej analizy kodu, Scala 3 oferuje API refleksji TASTy[22, 26]. TASTy jest binarnym formatem serializacji typowanych drzew składniowych używanym przez kompilator Scali 3[18].

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[22, 26].

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 poprzednim rozdziale do rozwiązania rzeczywistego problemu inżynierskiego: automatycznej generacji wydajnego analizatora leksykalnego z definicji wysokopoziomowej w formie języka dziedzinowego (DSL).

System `alpaca.lexer` implementuje transformację deklaracyjnych reguł tokenizacji zapisanych jako funkcja częściowa (ang. *partial function*) w kod proceduralny wykonywany w czasie kompilacji. Wykorzystuje przy tym pełne spektrum możliwości refleksji TASTy[22], włączając generację klas w czasie kompilacji, transformację drzew AST[26] oraz wyspecjalizowane typy `refinement`.

3.1.2. Architektura systemu leksera

Interfejs użytkownika

System oferuje użytkownikowi przejrzysty interfejs DSL oparty na dopasowaniu wzorców:

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

Listing 3.1: Definicja typu `LexerDefinition`

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

Główny punkt wejścia systemu stanowi metoda `lexer`:

```
1 transparent inline def lexer[Ctx <: LexerCtx](  
2   using Ctx withDefault LexerCtx.Default,
```

```

3 )(
4   inline rules: Ctx => LexerDefinition[Ctx],
5 )(using
6   copy: Copyable[Ctx],
7   betweenStages: BetweenStages[Ctx],
8 )(using inline
9   debugSettings: DebugSettings,
10 ): 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.

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[17, 23, 24].

3.1.3. Analiza drzewa składni abstrakcyjnej

Dekonstrukcja funkcji częściowej

Kluczowym krokiem implementacji jest ekstrakcja reguł z definicji funkcji częściowej:

```

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

```

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

Ten fragment kodu wykorzystuje dopasowanie wzorców w *quotes* do dekonstrukcji[23] 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 def replaceWithNewCtx(newCtx: Term) = new ReplaceRefs[quotes.type].apply(
2   (find = oldCtx.symbol, replace = newCtx),
3   (find = tree.symbol, replace = Select.unique(newCtx, "lastRawMatched")),
4 )

```

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[26].

3.1.5. Ekstrakcja i kompilacja wzorców

Funkcja `extractSimple`

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

```

1 def extractSimple(
2   ctxManipulation: Expr[CtxManipulation[Ctx]],
3 ): PartialFunction[Expr[ThisToken], List[Expr[ThisToken]]] =
4   case '{ Token.Ignored(using $ctx) } =>
5     // ...
6
7   case '{ type t <: ValidName; Token.apply[t](using $ctx) } =>
8     // ...
9
10  case '{ type t <: ValidName; Token.apply[t]($value: String)(using $ctx)
11    } if value.asTerm.symbol == tree.symbol =>
12    // ...
13
14  case '{ type t <: ValidName; Token.apply[t]($value: v)(using $ctx) } =>
15    // ...

```

Listing 3.5: Funkcja `extractSimple`: dopasowywanie definicji tokenów

Wykorzystuje ona dopasowanie wzorców w *quotes* z ekstraktorem typów[23], 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 leksykalnej, odpowiedzialny za ekstrakcję i walidację wzorców tokenów podczas ekspansji makra[24]. 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

Kluczowym mechanizmem implementacyjnym makra **lexer** jest programatyczna konstrukcja klasy anonimowej w czasie kompilacji[21]. Proces ten wykorzystuje API refleksji TASTy[22] 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 listę deklaracji członków klasy (pól i metod)

Definicja członków klasy

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

1. **Pola tokenów** — 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** — 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)

Mechanizm typów rafinowanych stanowi fundamentalną cechę systemu typów Scali umożliwiającą precyzyjne wyrażenie struktury typów w czasie kompilacji[27]. 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[26]:

```

1 definedTokens
2   .unsafeFoldLeft(TypeRepr.of[Tokenization[Ctx]]):
3     case (tpe, '{ $token: DefinedToken[name, Ctx, value] }') =>
4       Refinement(tpe, ValidName.from[name], token.asTerm.tpe)
5   .asType match
6   case '[refinedTpe] =>
7     val newCls =
8       Typed(New(TypeIdent(cls)).select(cls.primaryConstructor).appliedToNone,
9         TypeTree.of[refinedTpe])
10
11     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*):

```

1 Tokenization[Ctx] & {
2   val TOKEN1: DefinedToken["NAME1", Ctx, Type1]
3   val TOKEN2: DefinedToken["NAME2", Ctx, Type2]
4   ...
5 }
```

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`, który jest implementowany przez klasę generowaną przez makro. Mechanizm działania typów strukturalnych został szczegółowo opisany w dokumentacji Scali [27]. Aby mechanizm `Selectable` działał poprawnie ze strukturalnymi typami i nie wymagał refleksji, klasa generowana przez makro musi implementować `type Fields <: NamedTuple.AnyNamedTuple`[28]. W naszym podejściu makro generuje definicję `type Fields` zawierającą wszystkie zdefiniowane tokeny i ich typy, dzięki czemu:

- IDE oraz kompilator znają z góry dostępne pola i ich typy (pełne uzupełnianie i sprawdzanie typów),
- wywołanie `c.NAZWA` jest bezpieczne typowo mimo mechanizmu dynamicznego wyboru nazwy.

```

1 val fieldType = definedTokens
2   .unsafeFoldLeft[(Type[? <: Tuple], Type[? <:
3     Tuple]]((Type.of[EmptyTuple], Type.of[EmptyTuple])):
```

```

3   case (
4       ('[type names <: Tuple; names], '[type types <: Tuple; types]),
5       '{ $token: DefinedToken[name, Ctx, value] }',
6   ) =>
7       (Type.of[name *: names], Type.of[Token[name, Ctx, value] *: types])
8   .runtimeChecked
9   .match
10  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[17]. Dostęp do tokenów odbywa się poprzez bezpośrednie odwołanie do pola klasy, co po kompilacji do kodu bajtowego JVM[29] redukuje się do instrukcji `getField` — operacji o złożoności $O(1)$ bez żadnego narzutu pośrednictwa.

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[27]. 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[27] 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:

```

1 class SimpleLexer {
2   val tokens: Map[String, Token[?, ?, ?]] = Map(
3     "NUMBER" -> ...,
4     "PLUS" -> ...
5   )
6   def apply(name: String): Token[?, ?, ?] = tokens(name)
7 }

```

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:

```

1 class MyLexer extends Tokenization[DefaultGlobalCtx] {
2   val NUMBER = DefinedToken[...]
3   val PLUS = DefinedToken[...]
4   protected def compiled: Regex = "(?<token0>[0-9]+)|(?<token1>\\+)"
5   // ...
6 }

```

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** jest częścią API kompilatora do raportowania błędów w czasie kompilacji[24, 25]. Przerwanie kompilacji 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 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.

Bibliografia

- [1] M. E. Lesk i E. Schmidt. *Lex: A lexical analyzer generator*. T. 39. Bell Laboratories Murray Hill, NJ, 1975.
- [2] S. C. Johnson i in. *Yacc: Yet another compiler-compiler*. T. 32. Bell Laboratories Murray Hill, NJ, 1975.
- [3] D. Beazley. *PLY (Python Lex-Yacc)*. 2005. URL: <https://www.dabeaz.com/ply/ply.html> (term. wiz. 19.03.2025).
- [4] D. Beazley. *SLY (Sly Lex-Yacc)*. 2016. URL: <https://sly.readthedocs.io/en/latest/sly.html> (term. wiz. 19.03.2025).
- [5] D. Beazley. *SLY Github*. URL: <https://github.com/dabeaz/sly> (term. wiz. 19.03.2025).
- [6] T. Parr, P. Wells, R. Klaren, L. Craymer, J. Coker, S. Stanchfield, J. Mitchell i C. Flack. *What's ANTLR*. 2004.
- [7] A. Moors, F. Piessens i M. Odersky. „Parser combinators in Scala”. W: *CW Reports* (2008).
- [8] *scala-parser-combinators Getting Started*. URL: https://github.com/scala/scala-parser-combinators/blob/main/docs/Getting_Started.md (term. wiz. 04.04.2025).
- [9] J. Boyland i D. Spiewak. „Tool paper: ScalaBison recursive ascent-descent parser generator”. W: *Electronic Notes in Theoretical Computer Science* 253.7 (2010).
- [10] A. A. Myltsev. „Parboiled2: A macro-based approach for effective generators of parsing expressions grammars in Scala”. W: *arXiv preprint arXiv:1907.03436* (2019).
- [11] L. Haoyi. *sfscala.org: Li Haoyi, FastParse: Fast, Programmable, Modern Parser-Combinators in Scala*. 2015.
- [12] *FastParse Getting Started*. URL: <https://com-lihaoyi.github.io/fastparse/#GettingStarted> (term. wiz. 04.04.2025).
- [13] L. Haoyi. *FastParse. Fast, Modern Parser Combinators*. URL: <https://www.lihaoyi.com/post/slides/FastParse.pdf> (term. wiz. 18.04.2025).
- [14] *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).
- [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. 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](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] *Quoted Code / Macros in Scala 3*. URL: <https://docs.scala-lang.org/scala3/guides/macros/quotes.html> (term. wiz. 25.10.2025).
 - [24] *Macros. Scala 3 Reference*. URL: <https://docs.scala-lang.org/scala3/reference/metaprogramming/macros.html> (term. wiz. 25.10.2025).
 - [25] *Scala 3 Macros*. URL: <https://docs.scala-lang.org/scala3/guides/macros/macros.html> (term. wiz. 25.10.2025).
 - [26] *Reflection / Macros in Scala 3*. URL: <https://docs.scala-lang.org/scala3/guides/macros/reflection.html> (term. wiz. 25.10.2025).
 - [27] *Selectable / Macros in Scala 3*. URL: <https://docs.scala-lang.org/scala3/reference/changed-features/structural-types.html> (term. wiz. 27.11.2025).
 - [28] *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).
 - [29] 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, wykorzystujący bibliotekę <i>SLY</i> . . .	7
1.3	Fragment nie działającego kodu w Pythonie, wykorzystujący bibliotekę <i>SLY</i>	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: <code>transparent inline def lexer</code>	14
3.3	Dekonstrukcja funkcji częściowej (dopasowanie AST do <i>CaseDef</i>)	15
3.4	Zastąpienie referencji starego kontekstu nowymi (<i>ReplaceRefs</i>)	15
3.5	Funkcja <i>extractSimple</i> : dopasowywanie definicji tokenów	16
3.6	Rafinowanie typu wynikowego o pola tokenów	18
3.7	Wynikowy typ leksera	18
3.8	Tworzenie typu <i>Fields</i>	18
3.9	Podejście oparte na mapowaniu dynamicznym	20
3.10	Podejście oparte na jawnej definicji klasy	20