Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I INŻYNIERII BIOMEDYCZNEJ

KATEDRA INFORMATYKI STOSOWANEJ

Praca dyplomowa magisterska

Projekt języka programowania wspierającego przetwarzanie rozproszone na platformach heterogenicznych.

Design of a programming language with support for distributed computing on heterogenous platforms.

Autor: *Kajetan Rzepecki*
Kierunek studiów: *Informatyka*
Opiekun pracy: *dr inż. Piotr Matyasik*

Kraków, 2015

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie, i nie korzystałem ze źródeł innych niż wymienione w pracy.

Serdecznie dziękuję opiekunowi pracy za wsparcie merytoryczne oraz dobre rady edytorskie pomocne w tworzeniu pracy.

Spis treści

1. Wstęp 7
1.1. Motywacja pracy 8
1.2. Zawartość pracy 10
2. Język FOOF 11
2.1. Podstawowe typy danych 12
2.2. Funkcje 12
2.3. Kontynuacje 14
2.4. Obsługa błędów 15
2.5. Przetwarzanie współbieżne i rozproszone 16
2.6. Makra 17
2.7. System modułowy 18
2.8. Inżynieria wiedzy w języku 20
3. Kompilator języka FOOF 23
3.1. Architektura kompilatora 23
3.2. Parsowanie 24
3.3. Makroekspansja 25
3.4. Obsługa systemu modułowego 28
3.5. Transformacja *Continuation Passing Style* 28
3.6. Optymalizacja i generacja kodu 31
4. Środowisko uruchomieniowe języka 33
4.1. Architektura środowiska uruchomieniowego 33
4.2. Implementacja podstawowych typów danych 35
4.3. Implementacja kontynuacji 36
4.4. Implementacja obsługi błędów 37
4.5. Implementacja procesów 39
4.6. Harmonogramowanie procesów 40
4.7. Implementacja Modelu Aktorowego 40

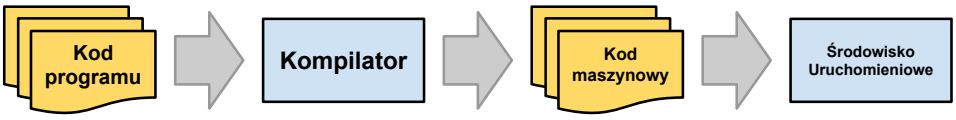
4.8. Dystrybucja obliczeń 42
5. Reprezentacja i przetwarzanie wiedzy 43
5.1. Reprezentacja wiedzy w języku 43
5.2. Algorytm Rete 43
5.3. Implementacja Rete - wnioskowanie w przód 44
5.4. Implementacja wnioskowania wstecz 44
5.5. Integracja z Systemem Uruchomieniowym 44
6. Podsumowanie 47
6.1. Kompilator języka FOOF 47
6.2. Środowisko uruchomieniowe 47
6.3. Przyszłe kierunki rozwoju 47
Bibliografia 49
A. Gramatyka języka FOOF 53
B. Przykładowe programy 55
B.1. Hello world! 55
B.2. Funkcja Fibonacciego 56
B.3. Obsługa błędów 56
B.4. Model Aktorowy 56
B.5. Współbieżne obliczenia funkcji Fibonacciego 57
B.6. System modułowy 58
B.7. Wnioskowanie w przód 59
B.8. Obsługa złożonych zdarzeń 60
B.9. Wnioskowanie wstecz 60
C. Spis wbudowanych funkcji i makr języka FOOF 63
D. Spisy rysunków i fragmentów kodu 65

1. Wstęp

Tematem pracy jest projekt i implementacja języka programowania wspierającego *przetwarzanie współbieżne* i umożliwiającego tworzenie *systemów rozproszonych* działających na *platformach heterogenicznych*.

Przetwarzanie współbieżne polega na podziale obliczeń na wiele procesów działających jednocześnie i konkurujących ze sobą o dostęp do ograniczonej ilości zasobów [1]. Procesy te mogą zostać rozproszone na wiele fizycznych maszyn, zachowując jednocześnie komunikację pomiędzy nimi, tworząc tym samym jeden koherentny system rozproszony [2].

Projektowany język programowania powinien więc udostępniać przejrzystą i ogólną notację umożliwiającą definiowanie komunikujących się procesów, jednocześnie pozostając językiem ogólnego przeznaczenia. Dodatkowym wymogiem jest prostota przy zachowaniu ekspresywności - język ten powinien być zbudowany w oparciu o niewielką liczbę ortogonalnych, dobrze współgrających ze sobą mechanizmów, które pozwalają na implementację szerokiej gamy funkcjonalności [3].



Rysunek 1.1: Schemat interakcji poszczególnych elementów języka.

W tym celu wymagane jest stworzenie kompilatora, czyli programu transformującego kod źródłowy języka programowania na format możliwy do uruchomienia w pewnym środowisku uruchomieniowym (ang. *runtime system*). Na środowisko takie zazwyczaj składa się zestaw podstawowych procedur wspólnych i niezbędnych do działania każdego programu. Schemat interakcji poszczególnych elementów projektu zaprezentowano na schemacie 1.1.

Ostatnim wymogiem postawionym przed projektowanym językiem, jest wyjście naprzeciw licznym problemom występującym w systemach rozproszonych, w szczególności problemowi *heterogeniczności*, co umotywowano w sekcji 1.1.

1.1. Motywacja pracy

Tworzenie systemów rozproszonych jest zadaniem bardzo trudnym i wymaga wykorzystania specjalnie do tego przeznaczonych narzędzi - języków programowania, systemów bazodanowych i infrastruktury sieciowej. Na przestrzeni lat zidentyfikowano wiele kluczowych problemów manifestujących się we wszystkich systemach rozproszonych niezależnie od ich przeznaczenia. Tanenbaum oraz Van Steen w [2] wymieniają następujące problemy:

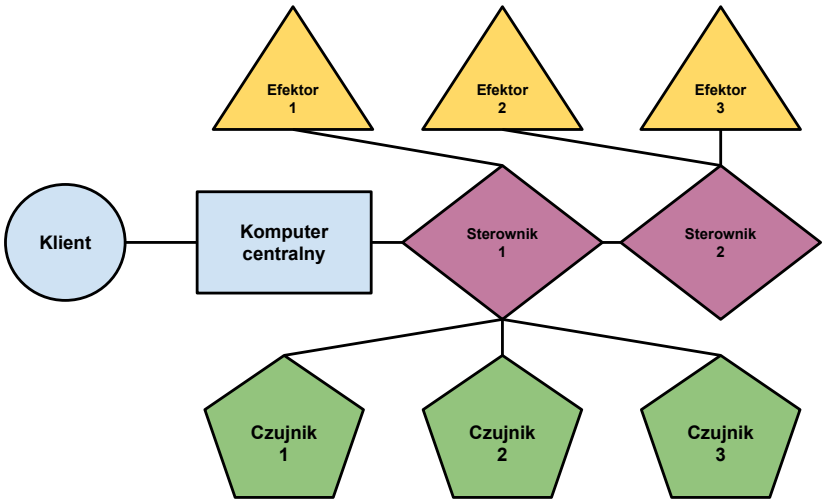
- **dostępność** - systemy rozproszone działają zwykle na wielu odrębnych maszynach, istotnym jest więc zachowanie dostępu do wspólnych zasobów z każdej części systemu,
- **przezroczystość dystrybucji** - istotnym jest również ukrycie fizycznego rozproszenia procesów i zasobów, dzięki czemu system rozproszony z zewnątrz stanowi jedną, koherentną całość,
- **otwartość** - polega na standaryzacji komunikacji pomiędzy poszczególnymi częściami systemu, dzięki czemu możliwe jest dodawanie nowych elementów bez ingerencji w pozostałe,
- **skalowalność** - systemy rozproszone powinny umożliwiać płynną zmianę ilości zasobów, być dostępne dla użytkowników z wielu lokacji geograficznych oraz umożliwiać łatwe zarządzanie niezależnie od ich rozmiaru.

Warto zauważyć, iż *skalowalność* jest przywoływana w folklorze programistycznym nieproporcjonalnie często, natomiast, istnieje wiele równie trudnych problemów, którym przeznacza się znacznie mniej uwagi, takich jak:

- **bezpieczeństwo systemu** - polega na kontroli dostępu do zasobów; w zależności od przeznaczenia systemu rozproszonego, jest najważniejszym aspektem jego budowy,
- **odporność na błędy** - polega na reagowaniu na zmiany (w szczególności wystąpienie błędów) zachodzące w systemie i podejmowaniu odpowiednich akcji w razie ich wystąpienia,
- **heterogeniczność** - polega na zróżnicowaniu platform sprzętowych wchodzących w skład fizycznej części systemu rozproszonego, a także poszczególnych logicznych części systemu.

Heterogeniczność jest problemem szczególnie trudnym, który powoli nabiera znaczenia wraz z pojawieniem się inicjatyw takich jak **Internet Rzeczy** (ang. *Internet of Things*) [4], gdzie systemy rozproszone zbudowane są z dużej ilości bardzo zróżnicowanych maszyn. Maszyny te cechują się różną architekturą sprzętową, ilością zasobów, a także przeznaczeniem i funkcjonalnościami, które realizują i umożliwiają.

Na schemacie 1.2 przedstawiony został przykład heterogeniczności platformy sprzętowej w kontekście Internetu Rzeczy. Klient, korzystając z centralnego komputera, uzyskuje dostęp do danych sensorycznych pochodzących z szerokiej gamy różnych czujników i bazując na ich wartości jest w stanie zmieniać zachowanie równie zróżnicowanych efektorów. Całość odbywa się za pośrednictwem dedykowanych sterowników, ułatwiających skalowanie systemu.

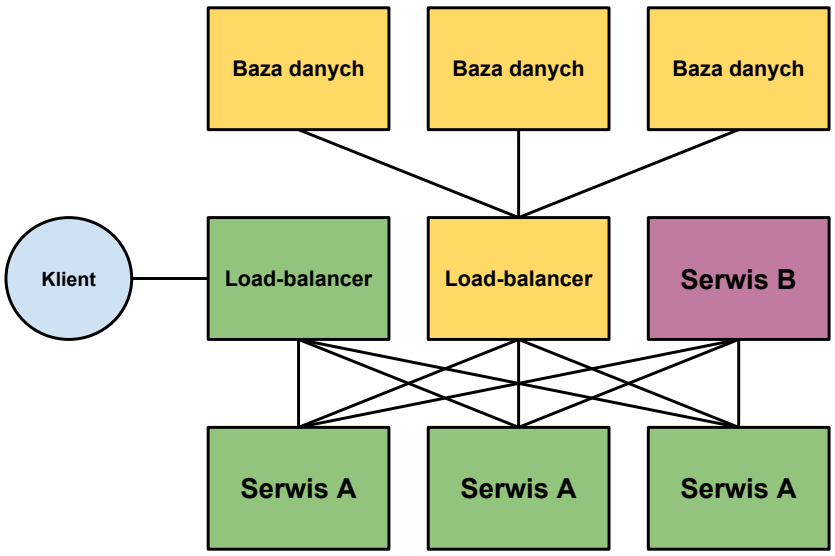


Rysunek 1.2: Przykład systemu opartego o heterogeniczną platformę sprzętową.

Każdy element takiego systemu, oznaczony na schemacie różnym kształtem oraz kolorem, reprezentuje maszynę udostępniającą różne zasoby i posiadającą różną fizyczną konstrukcję. Poszczególne elementy często działają w niekompatybilny sposób, w związku z czym wymagane jest wykorzystanie dedykowanych pośredników, których jedynym zadaniem jest *homogenizacja* systemu.

Problem heterogeniczności dotyka również systemów rozproszonych, które działają na platformach homogenicznych, gdzie fizyczne maszyny są do siebie bardzo zbliżone, a często są komputerami ogólnego przeznaczenia. Przykład takiego systemu, zbudowanego w oparciu o zdobywającą popularność architekturę mikroserwisową [5], został zawarty na schemacie 1.3.

Użytkownik systemu łączy się z głównym serwisem, który następnie komunikuje się z innymi serwisami, realizującymi wymagane przezeń zadania. W celu poprawienia *odporności na błędy* takiego systemu, w strategicznych miejscach umieszczono serwery zarządzające ruchem (ang. *load-balancer*), których zadaniem, analogicznie do przykładu ze schematu 1.2, jest *homogenizacja* systemu.



Rysunek 1.3: Przykład systemu heterogenicznego niezależnie od platformy sprzętowej.

W pierszym przypadku heterogeniczność wynika ze zróżnicowania maszyn należących do platformy sprzętowej, natomiast w drugim wynika ona z istnienia mikroserwisów, które realizują pojedyncze, konretnie sprecyzowane funkcjonalności. W obu przypadkach, heterogeniczność systemu prowadzi do powstania innych problemów, takich jak skalowalność i odporność na błędy, oraz konieczności wykorzystania dodatkowych elementów mających im zaradzić.

Często, sytuacja ta wynika z nieadekwatności narzędzi (w szczególności języków programowania) wykorzystanych do tworzenia systemu. Popularne języki programowania dążą do osiągnięcia **niezależności od platformy** (ang. *platform independence*) stosując maszyny wirtualne i inne techniki mające na celu homogenizację platformy sprzętowej, kiedy w rzeczywistości osiągają **ignorancję platformy** nie umożliwiając refleksji na jej temat.

Jako alternatywę dla osiągnięcia niezależności od platformy, niniejsza praca wprowadza termin **świadomości platformy** (ang. *platform awareness*), czyli dążenia do udostępnienia wiedzy o strukturze budowanego systemu rozproszonego oraz platformy sprzętowej, na której działa, i umożliwienia refleksji na jej podstawie. Zaprezentowany w dalszej części pracy język programowania, roboczo zwany `FOOF` [1], ma być uosobieniem ideologii świadomości platformy.

1.2. Zawartość pracy

– list what is found where in the thesis

[1] Nazwa pochodzi od difluorku ditlenu, niezwykle reaktywnego, dysruptywnego i niebezpiecznego związku chemicznego, który nie ma zastosowania.

2. Język FOOF

Niniejszy rozdział szczegółowo opisuje projekt języka programowania FOOF począwszy od podstawowych typów danych, przez notację funkcji, kontynuacji i procesów, kończąc na zaawansowanych mechanizmach języka, takich jak przetwarzanie wiedzy i wbudowany system makr. W dodatku A zawarto formalny opis gramatyki języka, natomiast w dodatku B zamieszczono kilka przykładowych programów.

Język FOOF został zaprojektowany bazując na cennych wskazówkach przedstawionych przez John'a Backus'a w wykładzie wygłoszonym przez niego podczas odbierania Nagrody Turing'a w 1977 roku [3]. Wskazówki te są ponadczasowe i stanowią dobrą podstawę do tworzenia języków programowania, a w dużym skrócie sprowadzają się do następujących punktów:

- **prostota lecz nie surowość** (ang. *simplicity, not crudeness*) - języki programowania powinny cechować się prostotą, lecz nie ograniczać ekspresywności programisty przez brak możliwości zrealizowania pewnych funkcjonalności, a co za tym idzie:

- **ortogonalne funkcjonalności** (ang. *orthogonal features*) - język programowania powinien składać się z niewielkiej liczby dobrze zdefiniowanych i dobrze współgrających mechanizmów, za pomocą których programista jest w stanie łatwo zbudować wszelkie inne potrzebne funkcjonalności.

Oczywiście, zasady te nie są wystarczające do stworzenia funkcjonalnego języka programowania, dlatego kierowano się także **pragmatyzmem**, który w kontekście projektowania języków programowania sprowadza się do podejmowania kompromisów, pomiędzy *matematyczną czystością* a faktyczną użytecznością potencjalnych funkcjonalności dostarczanych przez język. Podejście to zostało szczegółowo opisane w [6].

Ze względu na podobne zasady, którymi kierowano się podczas projektowania, język FOOF przypomina pod względem składniowym i semantycznym odpowiednio języki **Scheme** (opisany szczegółowo w [7]) oraz **Standard ML** [8]. Natomiast, cechami odróżniającymi FOOF od tych języków są: wsparcie dla programowania współbieżnego oraz wykorzystanie inżynierii wiedzy w celu osiągnięcia *świadomości platformy* i rozwiązania problemu heterogeniczności systemów rozproszonych.

2.1. Podstawowe typy danych

Listing 1 prezentuje proste typy danych dostępne w języku FOOF; są to podstawowe elementy budulcowe programów, które mają swoją reprezentację literałową.

```
23.5
symbol
:symbol
"ciąg znaków"
(1 2 3)
[1 2 3]
{:a 1 :b 2}
```

Listing 1: Podstawowe typy danych dostępne w języku FOOF.

Typy te to w kolejności: liczby, symbole, słowa kluczowe i ciągi znaków tekstowych, stanowiące wspólnie klasę wartości atomowych oraz listy pojedynczo-wiązane, wektory i mapy asocjacyjne. Każdy nieatomowy typ danych składa się z określonej liczby podwartości, które mogą być atomowe, lub nieatomowe. Semantyka każdego wymienionego typu danych jest zgodna z analogicznymi konstrukcjami opisanymi w [7].

Jako, że język FOOF jest dialektem języka Lisp, programy kodowane są homoikonicznie przez opisane powyżej typy danych - stosowana jest notacja **S-wyrażeń**, która została wprowadzona w [9]. Notacja ta rozmywa granicę pomiędzy programami a danymi, pozwalając programom na manipulację, budowę i transformację innych programów.

Homoikoniczność i notację S-wyrażeń wykorzystano w wielu innych mechanizmach dostępnych w języku, które zostały opisane w dalszej części niniejszego rozdziału, w szczególności w implementacji systemu makr pozwalającego na rozszerzenie składni języka.

2.2. Funkcje

Pierwszym złożonym typem danych, który nie ma reprezentacji literałowej w języku FOOF są funkcje. Funkcje są obiektami pierwszej klasy, to znaczy, po stworzeniu podczas działania programu, mogą być wykorzystywane tak jak każdy inny typ danych, a co za tym idzie, mogą być osadzane w listach, przekazywane do innych funkcji, a także z nich zwracane jako wynik obliczeń.

Funkcje zostały zaprojektowane w oparciu o **Rachunek Lambda**, wprowadzony w 1933 roku przez Alonzo Church'a jako alternatywny model logiki i, następnie, prowadzenia obliczeń [10, 11]. Rachunek ten wprowadza pojęcie **wyrażenia lambda**, które jest ekwiwalentem jednoargumentowych funkcji obecnych języków programowania, oraz szereg zasad substytucji,

zwanych redukcjami, pozwalających na uproszczenie zagnieżdżonych wyrażeń lambda. Najważniejszą z wprowadzanych redukcji jest β**-redukcja**, która konceptualnie reprezentuje aplikację funkcji z odpowiednimi argumentami i jednocześnie pozwala na prowadzenie obliczeń.

Zasady Rachunku Lambda są fundamentalnie bardzo nieskomplikowane, a mimo to pozwalają na ekspresję skomplikowanych idei, takich jak logika Bool'a, arytmetyka, struktury danych oraz rekurencja. Na listingu 2 zawarto przykład realizacji logiki boola wraz z kilkoma operatorami logicznymi w czystym Rachunku Lambda.

```
TRUE := λx.λy.x
FALSE := λx.λy.y

AND := λp.λq.p q p
OR := λp.λq.p p q
NOT := λp.λa.λb.p b a

AND TRUE FALSE
    ≡ (λp.λq.p q p) TRUE FALSE →β TRUE FALSE TRUE
    ≡ (λx.λy.x) FALSE TRUE →β FALSE
```

Listing 2: Przykład implementacji wartości i operatorów logicznych w Rachunku Lambda.

Wartości logiczne kodowane są jako wyrażenia lambda konsumujące dwa argumenty i wybierające odpowiednio pierwszy z nich, dla logicznej wartości prawdy, lub drugi z nich, dla logicznej wartości fałszu. W podobny sposób kodowane są operatory logiczne, a wynik ich działania obliczany jest przez sukcesywne przeprowadzanie substytucji nazwy argumentu na jego wartość oraz redukowaniu otrzymanych wyrażeń za pomocą β-redukcji.

Warto zauważyć, że wyrażenia lambda można interpretować jako tak zwane **domknięcia leksykalne**, czyli tworzone podczas β-redukcji otaczającego wyrażenia pary funkcji i map asocjacyjnych odzwierciedlających wartości zmiennych, które występują w ciele domknięcia leksykalnego, a nie są przez nie wprowadzane. Domknięcia leksykalne pozwalają opóźnić substytucję nazw argumentów wyrażeń lambda na odpowiadające im wartości, dzięki czemu są łatwiejsze w implementacji [12]. Listing 3 pokazuje działanie domknięć leksykalnych w notacji języka FOOF.

```
(let* ((x 23)
       (foo (lambda () x)))
  (let ((x 5))
    (display (foo)))) ;; Wyświetla liczbę 23
```

Listing 3: Przykład ilustrujący działanie domknięć leksykalnych.

Funkcja `foo` zaprezentowana na listingu, korzysta z wartości **wolnej zmiennej** `x`, czyli takiej, której nie wprowadza w liście swoich argumentów. W dalszej części programu, funkcja `foo`, pomimo lokalnej zmiany wartości zmiennej `x`, poprawnie zwraca oryginalną jej wartość, ponieważ w momencie jej tworzenia wartość zmiennej `x` została zapisana razem z ciałem funkcji.

Często pojawiającym się problemem związanym z funkcjami wzorowanymi na Rachunku Lambda, jest tak zwany problem **funarg**, polegający na niepoprawnym działaniu programów, które zwracają funkcje jako wynik obliczeń, lub przekazują je jako argumenty innych funcji. Problem ten sprowadza się do niewłaściwego budowania domknięć leksykalnych, co może doprowadzić do przedwczesnego usunięcia wartości zmiennych wolnych. Został on poruszony w [13].

Kolejnym problemem towarzyszącym funkcjom zrealizowanym jako domknięcia leksykalne jest nietrywialna implementacja rekurencji, wynikająca z ustalonej kolejności wykonywania działań - tworzenie domknięcia leksykalnego funkcji rekurencyjnej jest uzależnione od jej uprzedniego istnienia, co prowadzi do sprzeczności.

Oryginalna praca wprowadzająca Rachunek Lambda w celu osiągnięcia rekurencji wykorzystuje rachunek kombinatorów [10], a w szczególności **kombinator Y**. Sposób działania tego kombinatora został szczegółowo opisany w [14], natomiast problem i propozycję implementacji rekurencji szerzej opisano w [15].

2.3. Kontynuacje

Kolejnym mechanizmem będącym integralną częścią języka `FOOF` są kontynuacje, czyli abstrakcyjne reprezentacje przepływu sterowania programów, które pozwalają jednoznacznie określić kolejność wykonywania obliczeń.

Kontynuacje można interpretować jako ciąg obliczeń pozostałych do wykonania z punktu widzenia danego miejsca programu, który został **reifikowany** jako funkcja i udostępniony z poziomu wykonywanego programu. W efekcie, programy mogą zadecydować by zrestartować obliczenia od pewnego momentu, albo wręcz przeciwnie, przerwać je odrzucając wartości pośrednie.

Jako, że jest to mechanizm skomplikowany, który był odkrywany wielokrotnie [16], często nieświadomie, istnieje wiele jego wersji i sposobów implementacji, a w związu z czym nie jest on powszechnie dostępny jako standardowa funkcjonalność popularnych języków programowania. Ze względu na swoje właściwości opisane powyżej, kontynuacje są częściej stosowane w implementacjach kompilatorów języków programowania, jako format pośredni reprezentacji programów [17].

Języki programowania, które korzystają z kontynuacji czasem udostępniają je jako obiekty pierwszej klasy, które mogą być traktowane w taki sam sposób jak inne typy danych. Służy do

tego wiele zróżnicowanych operacji prymitywnych, które różnią się semantyką. W przypadku języka Scheme operacja prymitywna służąca do przechwytywania kontynuacji to `call-with-current-continuation` (`call/cc`) [7], natomiast w języku Standard ML służy ku temu konstrukcja `letcc` [18].

Istnieją także sposoby komponowania kontynuacji, bazujące na tak zwanych kontynuacjach ograniczonych (ang. *delimited continuations*), które wykorzystują większą liczbę operacji prymitywnych, na przykład `shitf` oraz `reset` opisane w [19], w celu zapewnienia większej kontroli nad przepływem sterowania programu. Listing 4 demonstruje sposób wykorzystania kontynuacji w języku `FOOF` w celu implementacji wczesnego powrotu z funkcji.

```
1 (lambda (x)
2   (letcc return
3     ...
4     (return 23)
5     ...))
```

Listing 4: Przykład wykorzystania kontynuacji w języku `FOOF`.

Dzięki możliwości przechwycenia kontynuacji, program jest w stanie przedwcześnie zakończyć działanie funkcji z obliczoną wartością. Kontynuacje dostępne są bezpośrednio, dzięki konstrtukcjom `letcc`, `shift` oraz `reset`, a także pośrednio, dzięki gamie innych mechanizmów kontroli przepływu sterowania, takich jak obsługa błędów, czy multiprocessing.

2.4. Obsługa błędów

Jednym z najważniejszych mechanizmów, jakie powinien udostępniać język programowania, jest mechanizm obsługi błędów i sytuacji wyjątkowych.

Język `FOOF` zapewnia mechanizm obsługi błędów, który bazuje na kontynuacjach, w związku z czym charakteryzuje się bardzo dużą ekspresywnością. Mechanizm ten umożliwia, analogicznie do większości popularnych języków programowania, zadeklarowanie procedury obsługi zdarzeń wyjątkowych za pomocą konstukcji `handle` oraz sygnalizację zajścia takiego zdarzenia poprzez `raise`.

W przeciwieństwie do większości języków programowania, mechanizm dostępny w języku `FOOF` pozwala na kontynuowanie obliczeń w miejscu wystąpienia błędu z nową wartością, obliczoną w zadeklarowanej procedurze obsługi błędu. Przykład ilustrujący taki schemat został zaprezentowany na listingu 5.

```
(handle (do ...
            (raise 'error) ;; Błąd w trakcie wykonywania obliczeń.
            ...)
        (lambda (error restart)
          ...
          (restart new-value))) ;; Kontynuacja z nową wartością.
```

Listing 5: Przykład wykorzystania mechanizmu obsługi błędów.

Przykładowy program deklaruje procedurę obsługi sytuacji wyjątkowej, a następnie przechodzi do kosztownych obliczeń, które przedwcześnie sygnalizują wystąpienie błędu. Przepływ sterowania zostaje przekazany do zadeklarowanej procedury obsługi sytuacji wyjątkowej, która decyduje się zrestartować obliczenia dostarczając im nową, poprawną wartość. Następnie, program wraca do punktu wystąpienia błędu i kontynuuje obliczenia wykorzystują nową, poprawną wartość.

2.5. Przetwarzanie współbieżne i rozproszone

Jednym z głównych założeń języka jest wsparcie dla przetwarzania współbieżnego i rozproszonego, dlatego istotnym jest, by abstrakcja to umożliwiająca była prosta, ekspresywna i wygodna w użyciu, ponieważ będzie stanowiła kluczowy element każdego programu, który powstanie w języku FOOF. Abstrakcją, która spełnia wszystkie te wymogi jest **Model Aktorowy** zaproponowany przez Carl'a Hewitt'a w 1973 roku [20] i rozszerzony o formalny opis semantyki przez Williama Clingera w roku 1981 [21].

Model Aktorowy bazuje na kilku prostych koncepcjach, takich jak podział programu na wiele działających współbieżnie procesów (aktorów), porozumiewających się poprzez przesyłanie wiadomości, na których podstawie mogą podejmować lokalne decyzje, tworzyć kolejne procesy, lub wysyłać kolejne wiadomości. Listing 6 prezentuje wszystkie operacje prymitywne udostępniane przez Model Aktorowy.

```
(send (spawn (lambda ()
               (sleep 1000)
               (send (recv) 'message)))
      (self))

(equal? (recv) 'message)
```

Listing 6: Przykład wykorzystania prymitywnych operacji Modelu Aktorowego w języku.

Program ten tworzy nowy proces korzystając z funkcji `spawn`, któremu natychmiastowo wysyła wiadomość za pośrednictwem funkcji `send`, zawierając w niej swój identyfikator `self`, po czym przechodzi do oczekiwania na odpowiedź wywołując funkcję `recv`. Tymczasem, nowopowstały proces zostaje uśpiony na 1000 milisekund (`sleep`) po czym odbiera przesłaną do niego wiadomość i odpowiada na nią wysyłając symbol `message`.

Interfejs ten jest bardzo zbliżony do interfejsu Modelu Aktorowego dostępnego w języku Erlang [22] i zaiste był na nim wzorowany. W odróżnieniu od języka Erlang, odbieranie wiadomości nie wykorzystuje dopasowywania wzorców bezpośrednio w prymitywnej operacji `recv`, lecz umożliwia jego osobną implementację. Podobnie, jak w przypadku języka Erlang, projekt przewiduje rozszerzenie listy prymitywnych operacji o identyfikację maszyn, na których działają procesy.

Implementacja Modelu Aktorowego w języku `FOOF` podobnie jak mechanizm obsługi błędów, została oparta o kontynuacje i zostanie opisana szczegółowo w następnych rozdziałach.

2.6. Makra

Prawdobodobnie najciekawszą funkcjonalnością języków z rodziny Lisp jest ich podejście do metaprogramowania i generowania kodu. Większość języków z tej rodziny wykorzystuje wersję systemu **makr**, który pozwala rozszerzać składnię języka i tworzyć dialekty domenowe (ang. *domain specific language*) w prosty i przystępny sposób. Język `FOOF` nie jest wyjątkiem i również został wyposażony w system makr.

Listing 7 prezentuje efekt działania **makroekspansji**, czyli substytucji wywołań makr na definicje ich ciał, na przykładzie kilku wbudowanych makr rozszerzających składnię języka `FOOF`.

```
;; Przed makroekspansją:
(and 23 42)



(let ((x 23))
  (display x))


`(4 is ,(* 2 2))
```

```
;; Po makroekspansji:
(if 23
    42
    false)

((lambda (x)
   (display x))
 23)

(list '4 'is (* 2 2))
```

Listing 7: Przykład działania systemu makr w języku `FOOF`.

Efektem makroekspansji jest powstanie semantycznie ekwiwalentnego kodu, który wykorzystuje tylko dobrze zdefiniowane konstrukcje składniowe języka. Warto zwrócić uwagę na

ostatni z przykładów, który prezentuje znaną z innych dialektów języka Lisp konstrukcję `quasiquote`. Konstrukcja ta umożliwia budowanie programów w łatwy, wizualny sposób bez konieczności samodzielnego budowania drzew programu z wykorzystaniem funkcji `cons`, `list` i pokrewnych. Szczegółowy opis działania `quasiquote` został zawarty w [23].

Systemy makr czesto borykają się z problemami **higieniczności** generowanego kodu. Problem ten ilustruje przykład z listingu 8.

```
(define-macro (unless c . b)
  `(if (not ,c)
       (begin ,@b)
       #void))

(let ((not identity))
  (unless #t
    (display "Hello world!")))
```

Listing 8: Przykład ilustrujący problem higieniczności systemu makr w języku Scheme.

Zdefiniowane zostaje makro `unless`, którego zadaniem jest uruchamianie pewnych obliczeń jedynie, gdy podany warunek nie jest spełniony. W tym celu makro korzysta z konstrukcji `if` oraz funkcji `not`, nie zachowując, niestety, higieniczności, czego dowodzi druga część przykładu - lokalna zmiana wartości zmiennej `not` na funkcję tożsamości powoduje niewłaściwe działanie makra `unless`.

Problem higieniczności jest problemem skomplikowanym i zazwyczaj jego rozwiązanie oznacza poświęcenie części funkcjonalności systemu makr, na przykład poprzez ograniczenie go do translacji szablonów [7], lub znacznego jego skomplikowania, przez konieczność wprowadzenia hierarchicznej refleksji makroekspansji [24]. Niestety, system makr języka `FOOF` pozostawia ten problem otwartym.

Alternatywnym podejściem do problemu metaprogramowania, o którym warto wspomnieć są **f-wyrażenia** (ang. *f-expressions*, *fexprs*), polegające na podziale funkcji na dwa fundamentalne kompotenty - aplikatywny, indukujący ewaluację argumentów oraz operatywny, analogiczny do substytucji nazw argumentów na ich wartości w wyrażeniach lambda Rachunku Lambda [25]. Podejście to drastycznie komplikuje kompilację kodu źródłowego, w związku z czym nie zostało wykorzystane w języku `FOOF`.

2.7. System modułowy

W celu umożliwienia podziału kodu źródłowego programów na logicznie związane części i ułatwienia zarządzania nimi, nowoczesne języki programowania często udostępniają systemy

modułowe, wraz z niezbędnymi do ich działania rozszerzeniami składniowymi.

Systemy takie, w zależności od języka programowania, na potrzeby którego zostały zaprojektowane, różnią się sposobem działania oraz ekspresywnością. Mniej skomplikowane systemy modułowe polegają na zwyczajnych podstawieniach tekstowych z opcjonalnym wsparciem dla *przestrzeni nazw* w celu uniknięcia konfliktów identyfikatorów, natomiast bardziej skomplikowane umożliwiają kontrolę dostępu oraz definiowanie zależności pomiędzy poszczególnymi modułami.

System modułowy wykorzystywany w języku `FOOF` jest zbliżony w swojej funkcjonalności do analogicznego systemu języka Standard ML [18], a właściwe jego modyfikacji opisanej w [26], gdzie mamy do czynienia ze **strukturami** wiążącymi ze sobą definicje funkcji i zmiennych, oraz **funktorami**, które pozwalają parametryzować struktury. Listing 9 prezentuje przykład wykorzystania systemu modułowego języka `FOOF`.

```
1  (structure A
2   (define (foo x)
3     (+ 23 x)))
4
5  (module (B a)
6    (define (bar)
7      (a.foo 5)))
8
9  (let ((b (B A)))
10   (display (b.bar))) ;; Wyświetla liczbę 28
```

Listing 9: Przykład wykorzystania systemu modułowego języka `FOOF`.

W przykładzie definiowana jest struktura `A` dostarczającą funkcję `foo` oraz moduł (odpowiednik funktora) `B` parametryzowany przez submoduł `a`. Następnie, tworzona jest instancja modułu `B`, wykorzystując zdefiniowaną uprzednio strukturę `A`, i używana w dalszej części programu. Korzystanie z sytemu modułowego jest ułatwione, dzięki specjalnej składni dostępu do zawartości modułu `module.member`.

System ten pozwala w łatwy sposób zarządzać zależnościami modułów - wystarczy zmienić parametr przekazany przy tworzeniu instancji modułu `B` na inną strukturę, bez modyfikacji jego definicji, by osiągnąć zamierzone cele. Funkcjonalność ta jest bardzo przydatna przy tworzeniu bibliotek programistycznych, które mogą być parametryzowane modułami służącymi do powszechnych zadań, takimi jak moduł do logowania, lub moduł zawierający parametry konfiguracji aplikacji. W efekcie, biblioteki te nie narzucają z góry implementacji modułów parametryzujących, dzięki czemu mogą być łatwiej zintegrowane z różnymi programami.

Zasadniczą wadą systemu modułowego w zaprezentowanej powyżej formie, jest konieczność istnienia osobnej fazy **linkowania**, czyli tworzenia instancji modułów. Problem ten został szczegółowo przeanalizowany w [27], skutkując stworzeniem notacji *interfejsów modułów* ułatwiającej automatyczną rezolucję zależności. Rozwiązanie to jest dalekie od doskonałego, toteż język FOOF stosuje inne podejście opisane w sekcji **??**.

2.8. Inżynieria wiedzy w języku

Ostatnią i zarazem najbardziej zaawansowaną funkcjonalnością języka FOOF jest jego wsparcie dla inżynierii wiedzy (ang. *knowledge engineering*), objawiające się umożliwieniem refleksji na podstawie pewnych *informacji*, które zostały odkryte podczas działania programów. Wspomniane, powiązane ze sobą logicznie informacje, czyli **wiedza**, mogą dotyczyć wielu różnych aspektów działania aplikacji i są w dużej mierze uzależnione od domeny rozwiązywanych problemów.

Istnieje wiele metod reprezentacji i przetwarzania wiedzy, które różnią się sposobem dostępu do zdobytych informacji, a co za tym idzie, stosownością do rozwiązywania danych klas problemów [28]. Dlatego też, wybór konkretnej reprezentacji i mechanizmu przetwarzania wiedzy w języku FOOF uzależniony jest od pragmatycznego jego zastosowania.

Wiodącym zadaniem inżynierii wiedzy w języku FOOF jest realizacja jednego z głównych założeń języka, czyli osiągnięcia **świadomości platformy** poprzez zdobycie i udostępnienie wiedzy o platformie sprzętowej i samej aplikacji na niej działającej. Wiedza ta ma stanowić bazę do podejmowania decyzji o rozwoju obliczeń prowadzonych w aplikacji, a także o samej strukturze systemu.

W założeniu ma to umożliwić automatyczną konfigurację i ewolucję rozproszonych aplikacji zbudowanych z wykorzystaniem języka FOOF. Na przykład, system inteligentnego domu, po wykryciu podłączenia w odpowiednim pomieszczeniu czujnika temperatury o wyższej dokładności pomiarów niż dotychczasowo dostępna, powinien bez modyfikacji programu, ani ingerencji jego użytkownika, zacząć z niego korzystać. Natomiast, w przypadku katastrofalnego błędu rzeczonego czujnika, system powinien wrócić do korzystania z poprzedniego czujnika.

Literatura związana z tą dziedziną nauki, która zarazem dotyczy systemów rozproszonych o wysokiej heterogeniczności, takich jak Internet Rzeczy, bardzo często wykorzystuje podejście ontologiczne [29, 28, 30]. Polega ono na budowie ontologii domenowej na potrzeby systemu, gdzie wiedza jest reprezentowana jako instancje i klasy obiektów powiązanych ze sobią pewnymi zależnościami. Ontologia ta może być następnie odpytywana, a działający w niej algorytm rozumowania (ang. *reasoner*) pozwala odkrywać nowe zależności pomiędzy obiektami.

Rozwiązane to wchodzi w konflikt z założeniami języka FOOF przez swoje skomplikowanie, relatywną restrykcyjność i statyczność bazy wiedzy oraz kosztowność obliczeniową, toteż, pomimo niewątpliwych zalet, nie mogło zostać wdrożone. Alternatywnym rozwiązaniem za-

stosowanym w języku jest podejście regułowe, polegające na reprezentacji wiedzy w formie **faktów** o nienarzuconej strukturze i przetwarzaniu tej wiedzy za pomocą **reguł** weryfikujących ową strukturę. Listing 10 pokazuje podstawowe operacje związane z inżynierią wiedzy dostępne w języku FOOF.

```
(whenever set-of-conditions
  (lambda (_)
    (retract! some-fact)
    (assert! another-fact)))
(signal! an-event)
```

Listing 10: Przykład wykorzystania prymitywnych operacji bazy wiedzy w języku.

Przykład ten definiuje jedną regułę bez zagłębiania się w szczegóły implementacji systemu regułowego języka FOOF. Reguła ta, bazując na spełnialności pewnego zbioru warunków `set-of-conditions` modyfikuje wbudowaną bazę faktów przez **asercję** i **retrakcję** rożnych faktów. Dodatkowo, przykład **sygnalizuje** zaistnienie pewnego zdarzenia `an-event`, co konceptualnie jest tożsame z asercją i późniejszą retrakcją faktu opisującego zajście tego zdarzenia w systemie.

Projekt przewiduje wykorzystanie opisanych powyżej podstawowych operacji oraz wiedzy możliwej do zdobycia podczas kompilacji programów języka FOOF poprzez **inferencję** faktów dotyczących struktury ich kodu źródłowego, do rozwiązania opisanego w sekcji 2.7 problemu linkowania modułów. Rozwiązanie to polega na przetworzaniu inferowanej wiedzy za pomocą zbioru reguł, zwanych kolektywnie **protokołem modułu**, w celu rezolucji zależności je spełniających w sposób automatyczny. Moduły, zamiast dokładnego sprecyzowania swoich zależności, mogą podać protokóły, które są konieczne i wystarczające do poprawnego, wspólnego działania, a system regułowy automatycznie wybierze spełniające je, dostępne submoduły.

3. Kompilator języka FOOF

Niniejszy rozdział przedstawia implementację kompilatora języka programowania FOOF szczegółowo opisując jego architekturę i poszczególne fazy kompilacji programów.

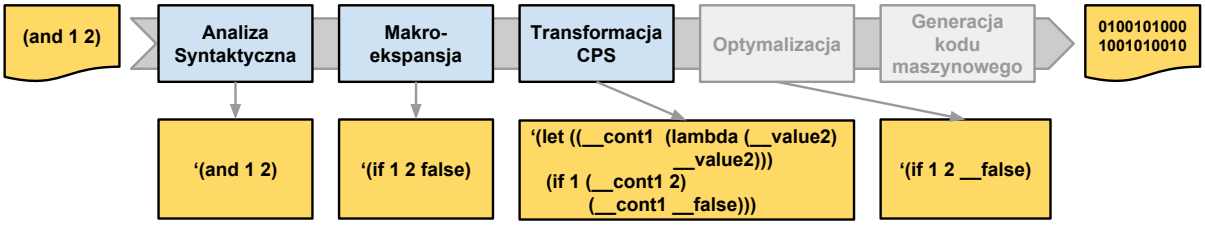
Kompilator jest programem komputerowym, którego głównym zadaniem jest **transformacja kodu źródłowego** programów do formatu bardziej odpowiedniego do uruchomienia przez maszynę [31]. Wynikiem działania kompilatora jest najczęściej plik wykonywalny zawierający instrukcje możliwe do uruchomienia przez procesor lub *maszynę wirtualną*, w przypadku kompilatorów *kodu bajtowego*. Kompilacja jest zwykle podzielona na kilka osobnych faz, takich jak analiza leksykalna, analiza semantyczna, optymalizacja i generacja kodu, które różnią się typem i celem przeprowadzanych transformacji [31].

Kompilator języka FOOF jest w założeniu kompilatorem *kodu maszynowego*, wynikiem działania którego jest strumień instrukcji możliwych do uruchomienia przez procesor komputera. Niestety, w wyniku ograniczeń czasowych i samej wielkości takiego projektu, ostatnie fazy kompilacji odpowiedzialne za generację kodu maszynowego zostały pominięte, a kompilacja programów kończy się na uruchamialnej reprezentacji pośredniej programów.

Towarzyszący pracy program został napisany w sposób inkrementalny [32] z wykorzystaniem wspólnego podzbioru języków Scheme oraz FOOF, celem późniejszego osiągnięcia autokompilacji (ang. *bootstrapping compiler*). Podczas tworzenia kompilatora nieocenione okazały się wskazówki na temat implementacji *języków funkcyjnych*, do których należy również język FOOF, przedstawione w [12].

3.1. Architektura kompilatora

Architektura kompilatora języka FOOF jest typowa dla tego typu programów [31], kompilacja została podzielona na jeden przebieg, na który składa się kilka logicznie po sobie następujących faz. Schemat 3.1 prezentuje obecnie zaimplementowane w kompilatorze fazy oraz te, których implementacja została przewidziana w przyszłości, wraz z przykładami pośrednich reprezentacji programów w nich występujących.



Rysunek 3.1: Schemat poszczególnych faz kompilacji i przykładowych danych będących wynikiem ich działania.

Pierwszą fazą jest faza **analizy leksykalnej i syntaktycznej** polegająca na transformacji kodu źródłowego - tekstu enkodującego programy - do formatu wewnętrznego możliwego do przetworzenia przez następne fazy kompilacji. Analiza syntaktyczna wykorzystuje opisaną w rozdziale 2 homoikoniczność języka `FOOF` i reprezentuje programy jako drzewa zbudowane z podstawowych typów danych dostarczanych przez język.

Druga faza kompilacji to faza **makroekspansji** polegająca na uproszczeniu konstrukcji syntaktycznych występujących w programach za pomocą szeregu transformacji. Faza ta pozwala uprościć analizę semantyczną pojawiającą się w późniejszych fazach kompilacji dzięki redukcji liczby różnych konstrukcji języka, które muszą być brane pod uwagę.

Trzecią fazą jest faza **konwersji *Continuation Passing Style*** polegająca na syntaktycznej transformacji kodu źródłowego programów celem wplecenia do niego **kontynuacji** [17]. Reprezentacja pośrednia programów po tej fazie kompilacji różni się zasadniczo od dotychczasowej reprezentacji, dzięki czemu ułatwia implementację szeregu opisanych wcześniej funkcjonalności języka.

Ostatnie dwie fazy kompilacji, czyli fazy **optymalizacji** i **generacji kodu maszynowego** w dużej mierze polegają na uproszczeniu przetworzonego kodu programów i przetłumaczeniu go na strumień prostych instrukcji możliwych do uruchomienia przez procesor komputera.

Kompilator działa w pojedynczym przebiegu, podczas którego każda z faz jest uruchamiana dokładnie jeden raz. Szczegółowy opis działania i implementacji poszczególnych faz został zawarty w dalszej części rozdziału.

3.2. Parsowanie

Pierwszym logicznym elementem kompilatora jest **parser** przeprowadzający analizę leksykalną i syntaktyczną. Jego zadaniem jest transformacja tekstu reprezentującego programy języka `FOOF` do drzewiastej reprezentacji bazującej na podstawowych typach danych udostępnianych przez język.

W związku z wyborem technologii wykorzystanych do implementacji kompilatora, budowa parsera przez niego używanego została oparta o, będące częścią standardu języka Scheme, pod-

stawowe funkcje operujące na plikach i kodzie źródłowym, takie jak `with-input-from-file` oraz `read` [7]. Implementacja dedykowanych parserów jest żmudna i nie prezentuje zbyt wysokiej wartości poznawczej, natomiast tworzenie generatora parserów, pomimo niewątpliwej ciekawości i przydatności z punktu widzenia użyteczności języka programowania, nie stanowi tematu niniejszej pracy. Powstała implementacja jest więc minimalną wersją niezbędną do umożliwienia dalszej kompilacji programów.

Opisany parser wspiera podstawową składnię języka FOOF wynikającą z jego homoikoniczności oraz jedno rozszerzenie składniowe usprawniające generowanie programów wewnątrz kompilatora. Rozszerzenie to polega na transformacji kombinacji znaków specjalnych `'`, `` ` ``, `,` oraz `,@` do odpowiadających im konstrukcji w formacie S-wyrażeń. Listing 11 pokazuje kod źródłowy i kod powstały po ekspansji syntaktycznej przeprowadzonej przez parser.

```
;; Kod źródłowy:
'(some value)
`(a ,b ,@c)
```

```
;; Po ekspansji:
(quote (some value))
(quasiquote (a (unquote b)
                (unquote-splicing c)))
```

Listing 11: Obsługa rozszerzeń składniowych S-wyrażeń w języku FOOF.

Należy zauważyć, że ekspansja syntaktyczna powyższego rozszerzenia składniowego nie jest jednoznacza z ekspansją wynikającą z jego znaczenia, która jest przeprowadzana w fazie makroekspansji opisanej w sekcji 3.3.

Naturalnie, istnieje możliwość łatwej wymiany implementacji parsera w przyszłości na mechanizm bardziej rozbudowany, wspierający dowolną ilość rozszerzeń składniowych. Można do tego celu wykorzystać generator parserów zbudowany w oparciu o gramatyki PEG (ang. *Parsing Expression Grammars*) [33] oraz monadyczne kombinatory parserów [34], które szczególnie dobrze nadają się do budowy parserów dla języków o nieskomplikowanej podstawowej gramatyce, takich jak FOOF.

3.3. Makroekspansja

Pierwszą fazą transformującą w znaczący sposob kod źródłowy języka FOOF, jest faza makroekspansji. Jej głównym zadaniem jest umożliwienie istnienia opisanego w rozdziale 2 systemu makr, a polega ona na aplikacji definicji makr do fragmentów kodu źródłowego znajdujących się w miejscu ich wywołań. Listing 12 prezentuje pseudokod algorytmu realizującego fazę makroekspansji w notacji języka FOOF.

```
(define (macroexpand expression defined-macros)
  (if (and (list? expression)
           (not (quote? expression)))
      (if (macro-defined? (macro-name expression) macros)
          (macroexpand (apply-expander (macro-name expression)
                                       defined-macros
                                       expression)
                       defined-macros)
          (map (lambda (subexpression)
                 (macroexpand subexpression macros))
               expression))
      expression))
```

Listing 12: Pseudokod algorytmu makroekspansji w notacji języka FOOF.

Algorytm ten jest rekurencyjny i przebiega w następujący sposób:

– Jeśli wyrażenie jest listą, której pierwszy element jest nazwą zdefiniowanego uprzednio makra, to następuje makroekspansja wyrażenia otrzymanego przez aplikację definicji makra do owego wyrażenia.

– Jeśli wyrażenie jest listą, ale jej pierwzy element nie identyfikuje zdefiniowanego uprzednio makra, to następuje makroekspansja każdego podwyrażenia wchodzącego w skład tego wyrażenia.

– Jeśli wyrażenie nie jest listą to zostaje zwrócone bez zmian.

Powyższy algortym uwzględnia możliwość, w której analizowanym wyrażeniem jest specjalna formuła `(quote ...)` przerywająca makroekspansję. Podobnie jak w przypadku wyrażeń nie będących listami, formuła `(quote ...)` zostaje zwrócona bez zmian. Algorytm uwzględnia również sytuację, w której wynikiem ekspansji jednego makra jest wywołanie innego makra, dzięki rekurencyjnemu wywołaniu makroekspansji po aplikacji definicji makra. Sytuację tę obrazuje listing 13.

W przykładzie została wykorzystana konstrukcja `let*`, która semantycznie oznacza zagnieżdżoną deklarację zmiennych lokalnych `a` oraz `b`. Makro odpowiedzialne za ekspansję konstrukcji `let*` skutkuje wygenerowaniem dwóch wywołań makra `let`, które następnie jest zamieniane na wywołania funkcji anonimowych (tworzonych za pomocą konstrukcji `lambda`) z odpowiednimi parametrami.

```
;; Kod źródłowy:
(let* ((a 23)
       (b (+ a 5)))
  (* 2 b))

;; Pierwszy krok makroekspansji:
(let ((a 23))
  (let ((b (+ a 5)))
    (* 2 b)))

;; Drugi krok makroekspansji:
((lambda (a)
   (let ((b (+ a 5)))
     (* 2 b)))
 23)

;; Trzeci krok makroekspansji:
((lambda (a)
   ((lambda (b)
      (* 2 b))
    (+ a 5)))
 23)
```

Listing 13: Przykład działania algorytmu makroekspansji.

Obecna implementacja nie wspiera definiowania nowych makr przez użytkowników języka FOOF. Powodem tej niedogodności jest nietrywialna interakcja systemu makr i systemu modułów zastosowanego w języku, która została szczegółowo przeanalizowana w [27]. System makr do poprawnego funkcjonowania wymaga znajomości definicji makr, które znajdują się w różnych modułach, przed uruchomieniem programu, natomiast system modułowy wymaga uruchomienia programu w celu przeprowadzenia linkowania modułów. Rozwiązanie tego problemu jest nietrywialne, w związku z czym nie zostało uwzględnione w projekcie języka. Lista predefiniowanych makr dostępnych w języku FOOF została zawarta w dodatku C.

Innym problemem manifestującym się w wielu systemach makr jest opisany w sekcji 2.6 problem higieniczności, polegający na nieoczekiwanej injekcji nieprawidłowych wartości do kodu generowanego przez makra. W zwązku z opisaną powyżej niedogodnością, problem ten nie jest obecny w implementacji języka FOOF i jego rozwiązanie stanowi problem otwarty. Znanych jest kilka sposobów rozwiązania problemu higieniczności systemu makr, na przykład wykorzystanie specjalnego systemu typów [35], lub wieży refleksji makroekspansji [24].

3.4. Obsługa systemu modułowego

Implementacja systemu modułowego języka FOOF wymaga niewielkiego wsparcia, w chwili obecnej, ze strony kompilatora.

Możliwość definiowania modułów została zreazilowana z wykorzystaniem systemu makr jako wywołania makr `structure` oraz `module` odpowiadające odpowiednio strukturom i funktorom opisanym w sekcji 2.7. Makra te generują wywołania specjalnej funkcji `&make-structure` budującej struktury z prostych wartości. Listing 14 prezentuje wynik makroekspansji makra `module`, której pośrednim krokiem jest ekspansja makra `structure`.

```
;; Kod źródłowy:
(module (X a b)
  (define (foo x)
    ...)
  (define (bar y)
    ...))
```

```
;; Po makroekspansji:
(define (X a b)
  (letrec ((foo (lambda (x)
                  ...))
           (bar (lambda (y)
                  ...)))
    (&make-structure
     'foo foo
     'bar bar)))
```

Listing 14: Przykład ekspansji makra `module`.

Definicje należące do zdefiniowanego modułu `X` transformowane są do postaci wzajemnie rekurencyjnej za sprawą konstrukcji `letrec`, a następnie ich wartości łączone są w jeden obiekt struktury. Do definicji należących do tak otrzymanej wartości można odnosić się z wykorzystaniem specjalnej składni `module.member`, która została zrealizowana jako ekspansja symboli w fastępnej fazie kompilacji opisanej w sekcji 3.5. Wywołanie funkcji `bar` instancji `x` modułu `X` wygląda więc następująco: `(x.bar 23)`.

3.5. Transformacja *Continuation Passing Style*

Kolejną fazą kompilacji jest faza konwersji przekazywania kontynuacji (ang. *Continuation Passing Style*, *CPS*) polegająca na automatycznej transformacji kodu źródłowego programu do formatu, w którym wszystkie funkcje przyjmują dodatkowy argument będący sukcesywnie przekazywaną dalej kontynuacją [17].

Celem tej fazy jest wplecenie notacji kontynuacji opisanych w sekcji 2.3 do pośredniej reprezentacji programów. Listing 15 prezentuje przykład konwersji CPS prostej funkcji.

```
;; Styl bespośredni:
(lambda (x y)
  (* 2 (+ x y)))
```

```
;; Styl Continuation Passing:
(lambda (x y cont)
  (__+ x y
       (lambda (v)
         (__* 2 v cont))))

;; Konwersja wbudowanych funkcji:
(define (__+ a b cont)
  (cont (+ a b)))
```

Listing 15: Przykład konwersji *Continuation Passing Style*.

Po transformacji, funkcja ta przyjmuje dodatkowy argument `cont`, który następnie przekazuje dalej w ciągu obliczeń. Analogicznie, wbudowane funkcje dodawania `+` i mnożenia * również przyjmują dodatkowy argument, który wywołują z wynikiem odpowiedniej operacji, powodując aplikację kontynuacji.

W przykładzie można zauważyć doprecyzowanie kolejności wykonywania działań po transformacji CPS - pierwszą wykonaną operacją jest dodawanie, a jego wynik przekazywany jest do, specjalnie w tym celu stworzonej, kontynuacji pośredniej i następnie od operacji mnożenia wraz z kontynuacją `cont` wywołania funkcji.

Algorytm automatycznej konwersji CPS polega na analizie struktury kodu źródłowego metodą *dziel i zwyciężaj* i przeprowadzeniu serii prostych podstawień, z których najważniejsze to:

– Transformacja identyfikatorów przebiega przez dodanie prefixu `__` i unormowanie znaków specjalnych w celu wyraźnego odseparowania wartości przed i po konwersji.

– Transformacja wartości prostych polega na wywołaniu *aktualnej kontynuacji* z ich wartością.

– Transformacja funkcji polega na rozszerzeniu listy ich argumentów o dodatkowy argument reprezentujący *kontynuację wywołania funkcji* i rekurencyjnym przeprowadzeniu transformacji ciała funkcji przy jednoczesnej podmianie aktualnej kontynuacji na wprowadzoną uprzednio kontynuację wywołania funkcji.

Dokłady opis algorytmu konwersji *Continuation Passing Style* zawarto w [17]. Ponieważ konwersja CPS ma miejsce podczas kompilacji i przed uruchomieniem programu, toteż wartość *aktualnej kontynuacji* nie jest ustalona. W związku z tym, powyższy **algorytm generuje kod**, który będzie się składał na faktyczną wartość aktualnej podczas uruchomienia programu kontynuacji.

Jedną z cech konwersji CPS jest dokładne sprecyzowanie kolejności zachodzenia operacji w transfomowanych programach, co uwydatnia problem implementacji rekurencji. Zgodnie z

opisem problemu z sekcji 2.3, funkcje rekurencyjne (a także funkcje wzajemnie-rekurencyjne) wymagają istnienia własnej (pośrednio w przypadku funkcji wzajemnie-rekurencyjnych) wartości zanim będą mogły zostać zbudowane, co prowadzi do powstania sprzeczności. Nie jest to jednak do końca prawdziwe stwierdzenie, otóż funkcje rekurencyjne wymagają pewnej **lokacji**, w której znajdzie się ich wartość, podczas budowy tejże wartości, dzięki czemu ich implementacja jest możliwa. Listing 16 prezentuje wynik transformacji CPS konstrukcji `letrec` służącej do definiowania wzajemnie-rekurencyjnych funkcji.

```
(letrec ((even? (lambda (x)
                  ...
                  odd?
                  ...))
         (odd? (lambda (x)
                 ...
                 even?
                 ...)))
  (even? 7))
```

```
(let ((__even? nil)
      (__odd? nil))
  (set! __even? (lambda (x)
                  ...
                  __odd?
                  ...))
  (set! __odd? (lambda (x)
                 ...
                 __even?
                 ...))
  (__even? 7
           (lambda (value)
             value)))
```

Listing 16: Przykład transformacji konstrukcji letrec.

Transformacja CPS w tym przypadku umożliwia implementację funkcji rekurencyjnych tworząc dla ich wartości lokacje (`__even?` oraz `__odd?`), do których następnie wpisuje za pomocą konstrukcji `set!` zbudowane wartości. W efekcie, obie funkcje mają wszystkie niezbędne informacje i mogą korzystać z pozostałych funkcji wprowadzonych przez konstrukcję `letrec`. Rozwiązanie to jest analogiczne do techniki opisanej w [36] oraz stanowi preferowalną (pod warunkiem dopuszczenia istnienia mutacji w języku) alternatywę dla wykorzystania kombinatora `Y` [14], którego implementacja dla funkcji wzajemnie-rekurencyjnych jest nietrywialna. Innym podejściem do rozwiązania problemu rekurencji jest automatyczna eliminacja wzajemnej rekursji [37].

Warto zauważyć, iż proste modyfikacje podstawowego algorytmu konwersji *Continuation Passing Style*, polegające na generowaniu wywołań wbudowanych funkcji w stategicznych miejscach, mogą pomóc w implementacji szerokiej gamy mechanizmów kontroli przepływu sterowania, jakich jaki obsługa błędów oraz multiprocessing. Fakt ten został wdrożony do implementacji kompilatora języka `FOOF` i szczegółowo opisany w rozdziale 4.

3.6. Optymalizacja i generacja kodu

Ostatnie dwie fazy kompilacji to optymalizacja i generacja kodu wynikowego. Zadaniem tych faz jest uproszczenie, przyspieszenie i przygotowanie przetransformowanego w poprzednich fazach kodu do postaci możliwej do uruchomienia przez komputer.

Fazy te zostały niestety pominięte w związku z ich skomplikowaniem i ograniczeniami czasowymi nałożonymi na projekt. W chwili obecnej, kompilator języka `FOOF` kończy działanie produkując kod pośredni, będący uruchamialnym podzbiorem języków Scheme i `FOOF`, dzięki czemu może zostać uruchomiony przez interpretery i kompilatory tych języków.

W przyszłości istnieje możliwość relatywnie łatwego dodania pozostałych faz kompilacji. W szczególności, zaimplementowana już faza konwersji *Continuation Passing Style* opisana w sekcji 3.5 ułatwia implementację szerokiej gamy ciekawych optymalizacji, takich jak częściowa ewaluacja statycznych wartości (ang. *partial evaluation*), prowadząca do zwijania wartości stałych (ang. *constant folding*), oraz eliminacji jednakowych podwyrażeń (ang. *common subexpression elimination*) [38].

Implementacja fazy generacji kodu maszynowego wymagać będzie dodatkowo **konwersji domknięć leksykalnych** (ang. *closure conversion*) i opcjonalnie **lambda-unoszenia** (ang. *lambda lifting*) [12], których zadaniem jest przeniesienie definicji funkcji anonimowych wygenerowanego kodu do globalnej przestrzeni nazw. Listing 17 ilustruje działanie obu tych transformacji.

```
;; Oryginalny kod źródłowy:
(let* ((x 23)
       (plus-x (lambda (n)
                 (+ n x))))
  (plus-x 5))
```

```
;; Lambda-unoszenie:
(define (plus-x x n)
  (+ n x))

(let* ((x 23))
  (plus-x x 5))
```

```
;; Konwersja domknięć-leksykalnych:
(define __lambda0 (self n)
  (+ n (&value-of self 'x)))

(let* ((x 23)
       (plus-x (&closure __lambda0
                         'x x)))
  (&apply plus-x 5))
```

Listing 17: Przykład ilustrujący różnice pomiędzy algorytmami lambda-unoszenia oraz konwersji domknięć leksykalnych.

Konwersja domknięć leksykalnych polega na przeniesieniu definicji funkcji anonimowych do globalnej przestrzeni nazw oraz odpowiedniej modyfikacji miejsc tworzenia domknięć leksykalnych. Technika lambda-unoszenia, która zwykle jest wykonywana tuż po konwersji domknięć leksykalnych, polega na redukcji ilości stworzonych obiektów funkcyjnych przez promocję zmiennych wolnych domknięć leksykalnych do listy argumentów funkcji i modyfikacji miejsc wywołań funkcji w celu przekazania dodatkowych wartości. Technika ta pozwala ominać proces budowania domknięcia leksykalnego i jednocześnie przyspieszyć miejsca jego wywołań.

4. Środowisko uruchomieniowe języka

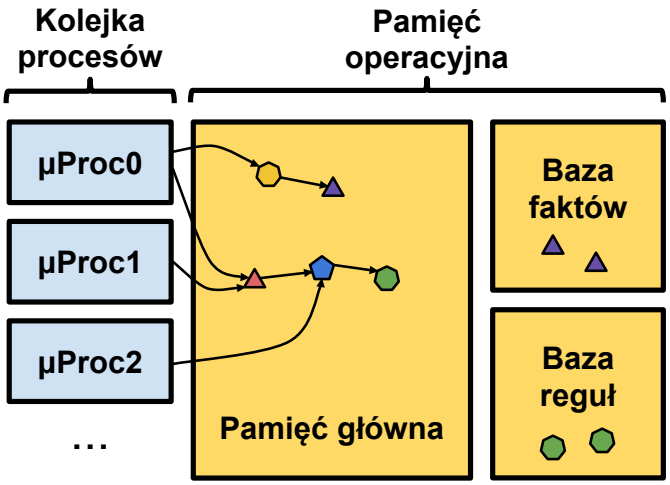
Niniejszy rozdział przedstawia architekturę środowiska uruchomieniowego (ang. *runtime system*) języka FOOF, czyli środowiska zawierającego procedury niezbędne do uruchamiania programów napisanych w tym języku. Do procedur takich należą te odpowiedzialne za budowę podstawowych typów danych dostarczanych przez język programowania, procedury zarządzania pamiecią programów, a także te niezbędne do działania zaawansowanych funkcjonalności języka.

W związku ze zróżnicowaniem funkcjonalności dostępnych w różnych językach programowania, nie istnieje jedna kanoniczna metoda implementacji ich środowisk uruchomieniowych. Sytuacja jest wręcz przeciwna, dwie różne implementacje tego samego języka programowania mogą posiadać zupełnie odmienne środowiska uruchomieniowe, natomiast dwa zupełnie różne języki mogą korzstać z tego samego środowiska uruchomieniowego, co często ma miejsce w przypadku *maszyn wirtualnych*. Przykładem takiej sytuacji jest wykorzystanie maszyny wirtualnej BEAM [22], oryginalnie zaprojektowanej dla języka Erlang, przez kilka innych języków programowania, takich jak Elixir i Joxa.

Język FOOF korzysta w obecnej postaci ze środowiska uruchomieniowego języka Scheme, rozszerzając jego funkcjonalność o mechanizmy niezbędne do implementacji przetwarzania współbieżnego, zaawansowanej obsługi błędów, a także zapewnienia wsparcia dla inżynierii wiedzy.

4.1. Architektura środowiska uruchomieniowego

Architektura środowiska uruchomieniowego wykorzystanego w implementacji języka FOOF jest relatywnie nieskomplikowa i składa się z niewielkiej liczby logicznych elementów. Po części jest to zasługa wiernego podążania za zasadami projektowania języków programowania przedstawionymi w [6] oraz wykorzystania środowiska uruchomieniowego języka Scheme. Diagram poszczególnych elementów logicznych i ich wzajemnej interakcji został zawarty na schemacie 4.1.



Rysunek 4.1: Schemat architektury środowiska uruchomieniowego języka FOOF.

Konceptualnie, pamięć dostępna dla środowiska uruchomieniowego języka FOOF została podzielona na dwa segmenty. Pierwszy z nich, oznaczony na diagramie kolorem niebieskim, zawiera jedynie struktury danych wykorzystywane przez środowisko uruchomieniowe, takie jak kolejka i deskryptory procesów działających w systemie, czy dane systemu uruchomieniowego języka Scheme. Drugi segment pamięci, oznaczony na diagramie kolorem żółtym, stanowi pamięć operacyjna, czyli pamięć przeznaczona i dostępna dla uruchamianych programów.

Segment pamięci operacyjnej został dodatkowo podzielony na trzy obszary, dwa z których zostały zarezerwowane na obsługę implementacji systemu regułowego do przechowywania baz faktów oraz reguł (rozdział 5), a trzeci, największy z nich, stanowi główny obszar, w którym przechowywane są obiekty reprezentujące wbudowane typy danych.

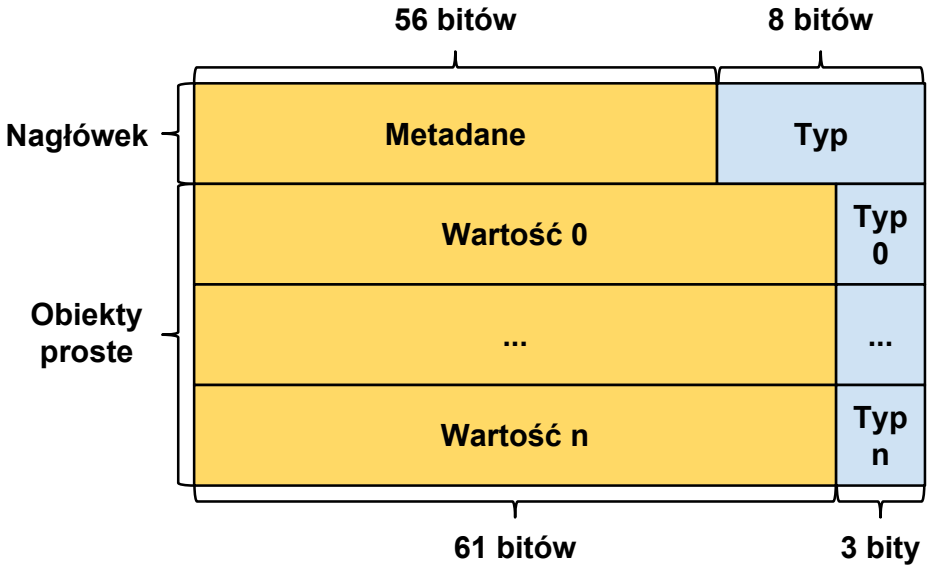
Główny obszar pamięci operacyjnej programów jest wspólny dla wszystkich procesów działających w systemie (symbolizowanych na diagramie przez bloki µProcN), dzięki czemu możliwe jest uniknięcie nadmiernego kompiowania danych podczas przesyłania wiadomości pomiędzy procesami kosztem synchronizacji dostępu do pamięci. Podejście to, zwane **stertą współdzieloną**, jest alternatywą do podejścia zastosowanego w implementacji języka Erlang, gdzie każdy proces działa w osobnej puli pamięci, przez co wymagane jest, często kosztowne, kompiowanie struktury wiadomości [22].

Obecna implementacja, w związku z ograniczeniami czasowymi narzuconymi na projekt, nie wykorzystuje potencjału przedstawionej architektury w stu procentach, ponieważ działa *jednowątkowo*, czyli jest ograniczona tylko do jednego wątku systemu operacyjnego. Nie ma to jednak wpływu na współbieżność procesów języka FOOF działających w środowisku uruchomieniowym, co zostało poruszone w sekcji 4.6.

W przyszłości istnieje możliwość rozwinięcia implementacji w celu wsparcia wielowątkowości, na przykład poprzez wykorzystanie **barier pamięci**, **operacji atomowych** oraz **pamięci lokalnej dla wątku** (ang. *thread-local storage*), co pozwoli osiągnąć przyspieszenie aplikacji języka `FOOF`.

4.2. Implementacja podstawowych typów danych

Wybór sposobu reprezentacji podstawowych typów danych w językach programowania jest bardzo ważny i często stanowi pole do czynienia kompromisów i optymalizacji [31]. Języki funkcyjne, do których należy język `FOOF`, z racji swojego nacisku na przejrzystą semantykę charakteryzują się relatywną prostotą reprezentacji wbudowanych typów danych [12]. Schemat 4.2 prezentuje przykład reprezentacji typów danych wprowadzony w [39] na potrzeby języków *dynamicznie typowanych*, czyli nie weryfikujących typów zmiennych podczas kompilacji.



Rysunek 4.2: Schemat przykładowej reprezentacji typów danych języków funkcyjnych.

Reprezentacja ta przewiduje istnienie dwóch klas obiektów:

– **prostych** - reprezentowanych przez jedno słowo procesora i posiadających krótki, trzybitowy tag określający ich dokładny typ,

– **złożonych** - reprezentowanych przez kilka kolejnych słów procesora, z których pierwsze zawiera nieco dłuższy, ośmiobitowy tag określający ich dokłady typ oraz pewien zestaw metadanych do wykorzystania przez środowisko uruchomieniowe, a następne są obiektami prostymi.

Reprezentacja taka pozwala w łatwy sposób enkodować wszystkie podstawowe typy danych języka FOOF. Na przykład, listy pojedynczo-wiązane mogą być zrealizowane jako ciąg par reprezentowanych przez obiekty złożone składające się z dwóch obiektów prostych - wskaźników wskazujących na inne obiekty złożone będące elementami pary. Integracja ze środowiskiem uruchomieniowym języka Scheme pozwoliła pominąć żmudną implementację reprezentacji i procedur konstrukcji wbudowanych typów danych języka. Ich semantyka jest więc zgodna z opisem zawartym w [7], a ekwiwalencję składniową zaprezentowano na listingu 18.

```
; Język FOOF:
23.5
symbol
:symbol
"ciąg znaków"
(1 2 3)
[1 2 3]
{:a 1 :b 2}
```

```
;; Język Scheme:
23.5
symbol
:symbol
"ciąg znaków"
(1 2 3)
#(1 2 3)
#hash((:a . 1) (:b . 2))
```

Listing 18: Porównanie wbudowanych typów danych języka FOOF i dialektu języka Scheme o nazwie Racket.

Wszystkie podstawowe typy danych języka FOOF mają swoje dokładne, semantyczne odpowiedniki w języku Scheme, a odróżnia je jedynie reprezentacja literałowa. Warto zauważyć, iż zależość ta jest prawdziwa także dla bardziej złożonych typów danych, jak funkcje, które są kodowane w ten sam sposób. Oba środowiska uruchomieniowe różną się natomiast reprezentacją kontynuacji - język FOOF używa konwersji przekazywania kontynuacji i reprezentuje je jako zwykłe funkcje - oraz brakiem idei procesów w standardzie języka Scheme.

4.3. Implementacja kontynuacji

Implementacja kontynuacji w języku FOOF została zrealizowana już podczas kompilacji za sprawą automatycznej konwersji *Continuation Passing Style*, która została szczegółowo opisana w sekcji 3.5.

Implementacja ta pozwala reprezentować kontynuacje za pomocą zwykłych funkcji, ale w celu ułatwienia implementacji pozostałych mechanizmów kontroli przepływu sterowania wymaga uwzględnienia pewnej modyfikacji. Modyfikacją tą jest wykorzystanie techniki **tampoliny**, polegającej na zwracaniu następnego kroku kontynuacji jako wyniku obecnego kroku zamiast bezpośredniego wywołania dalszej części kontynuacji [17].

Technika ta pozwala przerwać działanie kontynuacji pomiędzy poszczególnymi jej krokami przez zwyczajne nie-wywoływanie następnego kroku, a do jej implementacji wymagana jest je-

dynie modyfikacja kodu pośredniego programów w miejscach, w których normalnie następowałaby aplikacja następnej części kontynuacji. Listing 19 pokazuje efekt zastosowanej modyfikacji algorytmu konwersji CPS oraz wyników uruchomienia poszczególnych kroków kontynuacji.

```
;; Konwersja wbudowanych funkcji:
(define (__+ a b cont)
  (&yield-cont cont (+ a b)))

;; Przykładowe wyrażenie:
((lambda (x y cont)
   (__+ x y
        (lambda (v)
          (__* 2 v cont))))
 23
 5
 identity)
```

```
;; Po pierwszym kroku:
(&yield-cont (lambda (v)
               (__* 2 v cont))
             (+ 23 5))

;; Po drugim kroku:
(&yield-cont cont
             (* 2 28))

;; Po trzecim kroku:
56
```

Listing 19: Przykład uruchomienia funkcji z listing 15.

Przykład pokazuje, iż jedyną wymaganą modyfikacją jest emitowanie wywołań wbudowanej funkcji `&yield-cont` w miejscach bezpośredniego wywołania następnej kontynuacji. Funkcja `&yield-cont` zwyczajnie zwraca następny krok kontynuacji wraz z wartością, która ma do niego trafić - tak zwaną **dziurą kontynuacji**. W celu kompletnego uruchomienia kontynuacji należy sukcesywnie aplikować zwracaną funkcję do zwracanej wartości.

Wykorzystanie techniki trampoliny prowadzi do powstania *punktów sekwencyjnych* w programie, czyli zaistnienia miejsc, w których **gwarantowane** jest wykonanie dotychczasowych obliczeń (w szczególności *efektów* takich jak mutacja wartości). Miejsca te są analogiczne do punktów sekwencyjnych obecnych w interpreterach kodu bajtowego oraz interpreterach redukcyjnych [12], i mogą z powodzeniem służyć do realizowania podobnych funkcji - na przykład *debugowania*, lub *wywłaszczania*. Zostało to opisane w sekcji 4.6.

4.4. Implementacja obsługi błędów

Środowisko uruchomieniowe języka `FOOF` pozwala zrealizować obecny w języku zaawansowany mechanizm obsługi błędów i sytuacji wyjątkowych. Mechanizm ten został zaimplementowany w oparciu o kontynuacje i korzysta z dwóch funkcji wbudowanych dostarczanych przez środowisko uruchomieniowe.

Funkcje te, `&uproc-error-handler` oraz `&set-uproc-error-handler!` są odpowiedzialne za zarządzanie aktualnie aktywną procedurą obsługi sytuacji wyjątkowej, a ich wywoła-

nia są emitowane w fazie konwersji *Continuation Passing Style* opisanej szczegółowo w sekcji 3.5. Listing 20 demonstruje wykorzystanie wymienionych wyżej funkcji w generowanym w fazie konwersji CPS kodzie pośrednim.

```
;; Konwersja (raise error):
(let ((__handler (&uproc-error-handler)))
  (__handler __error
             (lambda (__value __ignored)
               (&set-uproc-error-handler! __handler)
               ...
               __value
               ...)))

;; Konwersja (handle expression new-handler):
(let ((__handler (&uproc-error-handler)))
  (&set-uproc-error-handler!
   (lambda (__error __restart)
     (&set-uproc-error-handler! __handler)
     ...
     __new-handler
     ...))
  ...
  __expression
  ...
  (&set-uproc-error-handler! __handler)
  ...)
```

Listing 20: Wykorzystanie kontynuacji do implementacji obsługi błędów.

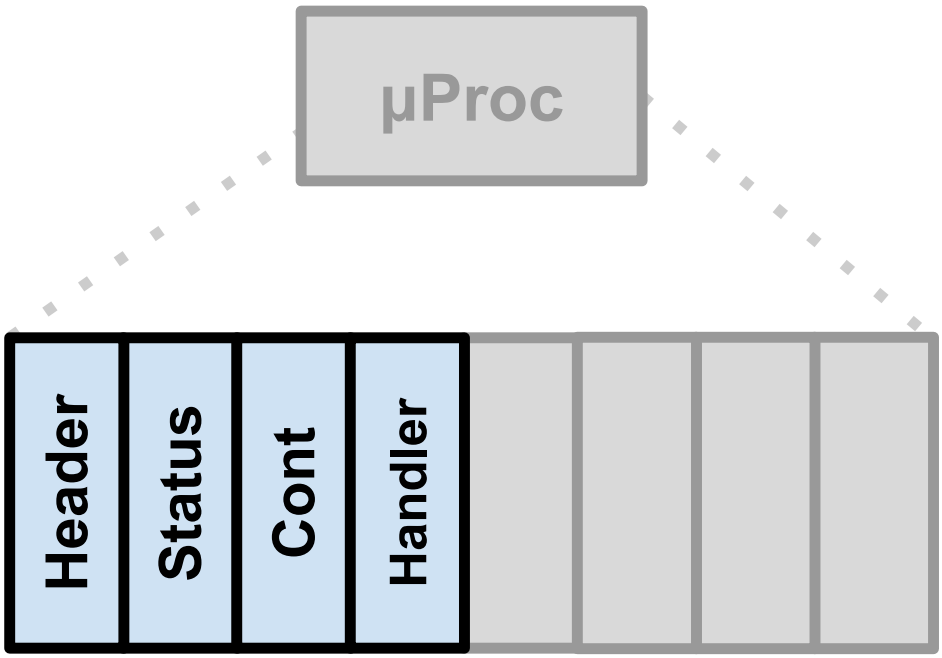
Powyższy, kryptyczny przykład pokazuje relatywnie skomplikowany kod generowany dla fundamentalnie nieskomplikowanych operacji `raise` oraz `handle` realizujących obsługę błędów w języku FOOF. Należy zwrócić uwagę na zastosowanie kontynuacji - procedura obsługi sytuacji wyjątkowej jest zwykłą, reifikowaną do postaci funkcji kontynuacją, której zadaniem jest wywołanie faktycznej funkcji obsługującej zdarzenie. Sygnalizacja zaistnienia błędu polega wtedy na aplikacji tak zbudowanej kontynuacji z sygnalizowaną wartością oraz aktualną kontynuacją, która realizuje restart obliczeń.

Skomplikowanie powyższego kodu wynika z konieczności odpowiedniego zarządzania procedurami obsługi zdarzeń wyjątkowych - musi istnieć gwarancja, że zrestartowane obliczenia zostaną uruchomione z tą samą procedurą obsługi zdarzeń, a po ich zakończeniu zostanie przy-

wrócona poprzednia procedura, natomiast nowodeklarowana procedura obsługi błędu powinna być uruchomiona w kontekście poprzedniej zadeklarowanej procedury. Alternatywnym rozwiązaniem, które prowadzi do nieznacznego zredukowania skomplikowania powyższego kodu, jest rozszerzenie bazowego algorytmu konwersji przekazywania kontynuacji w taki sosób, by przekazywane były dwie kontynuacje - pierwsza odpowiedzialna za prawidłowe wartości oraz druga, za obsługę sytuacji wyjątkowych [17]. Implementacja tego rozwiązania nie jest dostatecznie opłacalna, zatem nie została zrealizowana w implementacji języka `FOOF`.

4.5. Implementacja procesów

– add a diagram of the uProc context - only include status, cont & handler registers



Rysunek 4.3: Schemat kontekstu procesu obrazujący rejestry niezbędne do jego działania.

– describe uProc context registers

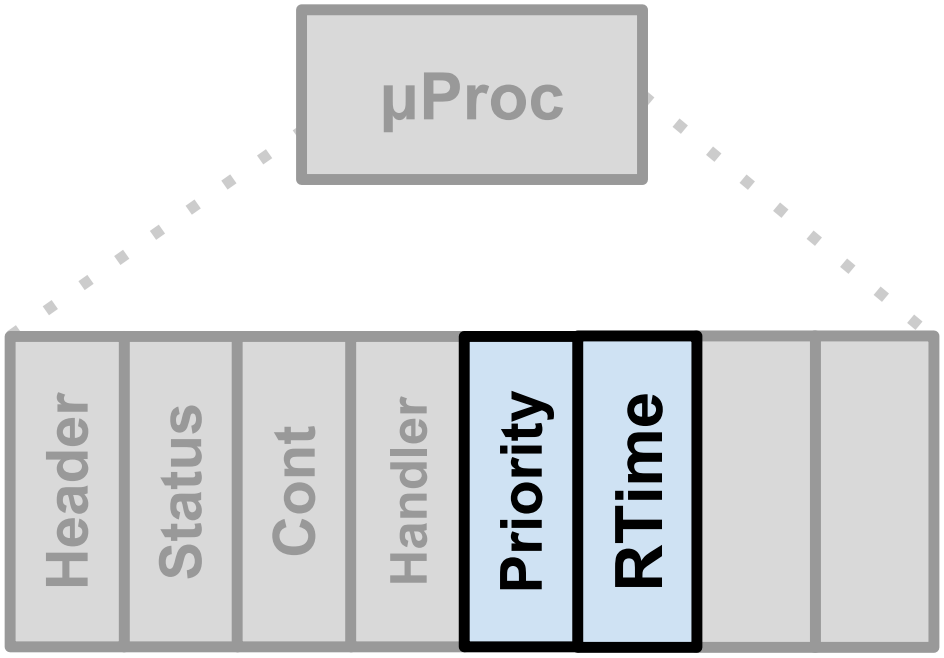
– describe how trampolines play into this scheme (recall `&yield-cont`)

– contrast trampolines with corutines (more suitable in CPS) and yielding (done implicitly) [[40]]

– describe how error handling is implemented (recall `&uproc-error-handler` etc)

– contrast with erlang [[22]]

4.6. Harmonogramowanie procesów

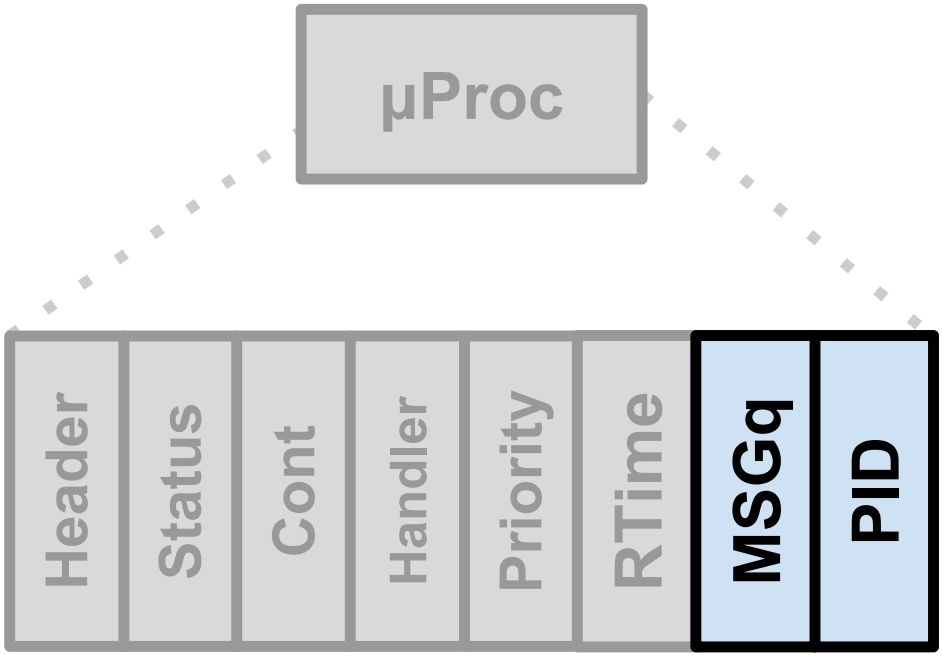


Rysunek 4.4: Dodatkowe rejestry kontekstu mikroprocesu wymagade do implementacji algorytmu *Completely Fair Scheduler*.

- describe the Completely Fair Scheduler [[41]]
- add pseudocode listing showing the algorithm
- describe uProc context switching
- contrast current impl with previous one (lack of wait list - notifications, heaps instead of RBT, number of reductions instead of time) [[42]]
- contrast with erlang [[22]]

4.7. Implementacja Modelu Aktorowego

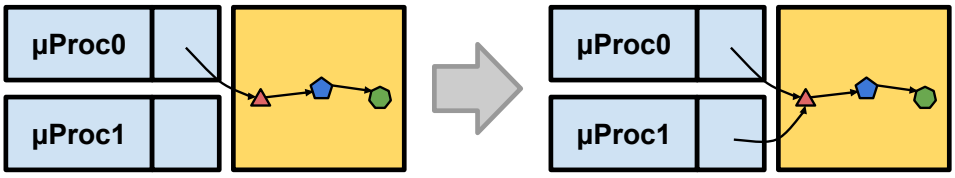
- describe actor model briefly [[20]] [[21]]



Rysunek 4.5: Dodatkowe rejestry kontekstu mikroprocesu wymagade do implementacji Modelu Aktorowego.

– describe modifications to the runtime required by actor model (**current-uproc**, uproc list, context fields)

– describe implementation of various actor model primitives



Rysunek 4.6: Diagram obrazujący efekty przekazywania wiadomości pomiędzy mikroprocesami.

– add some code examples and discussion of its effects and what happens

– contrast with erlang [[22]]

4.8. Dystrybucja obliczeń

– difference between concurrency & distribution

– describe modifications to the runtime in order to support distribution

– hint about using a simple protocol

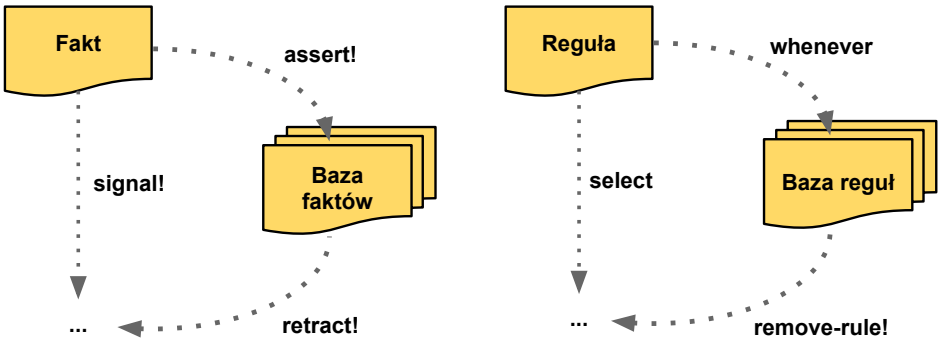
– hint about moving this into stdlib

5. Reprezentacja i przetwarzanie wiedzy

– describe how this needs a separate section

– elaborate on different ways of knowledge representation [[28]] [[43]] [[29]] [[?]] [[?]]

5.1. Reprezentacja wiedzy w języku



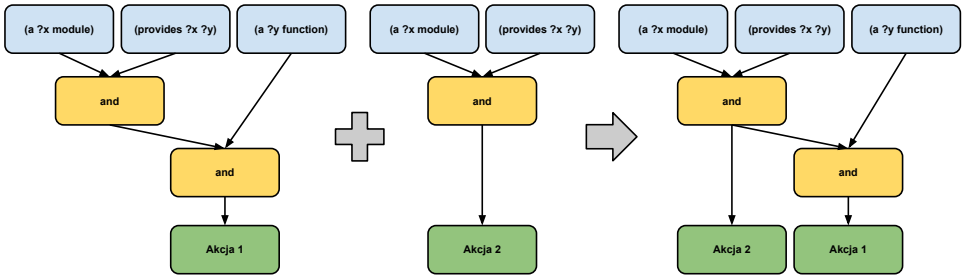
Rysunek 5.1: Schemat działania wbudowanych baz faktów i reguł.

– describe facts - signalling, assertion & retraction

– describe rules briefly - adding & disabling, triggering

5.2. Algorytm Rete

– describe in detail the algorithm [[44]]



Rysunek 5.2: Schemat łączenia podsieci w algorytmie *Rete*.

– describe briefly its history [[45]]

– Rete vs naïve approach (vs CLIPS or similar ?)

– add a benchmark diagram showing how Rete is better

– contrast it with other algorithms [[46]]

5.3. Implementacja Rete - wnioskowanie w przód

– describe what forward-chaining is

– describe naïve Rete - no network merging

– hint that this might be a good thing (future section)

– describe all the nodes [[44]]

5.4. Implementacja wnioskowania wstecz

– describe what backward-chaining is

– describe fact store in detail - linear, in-memory database

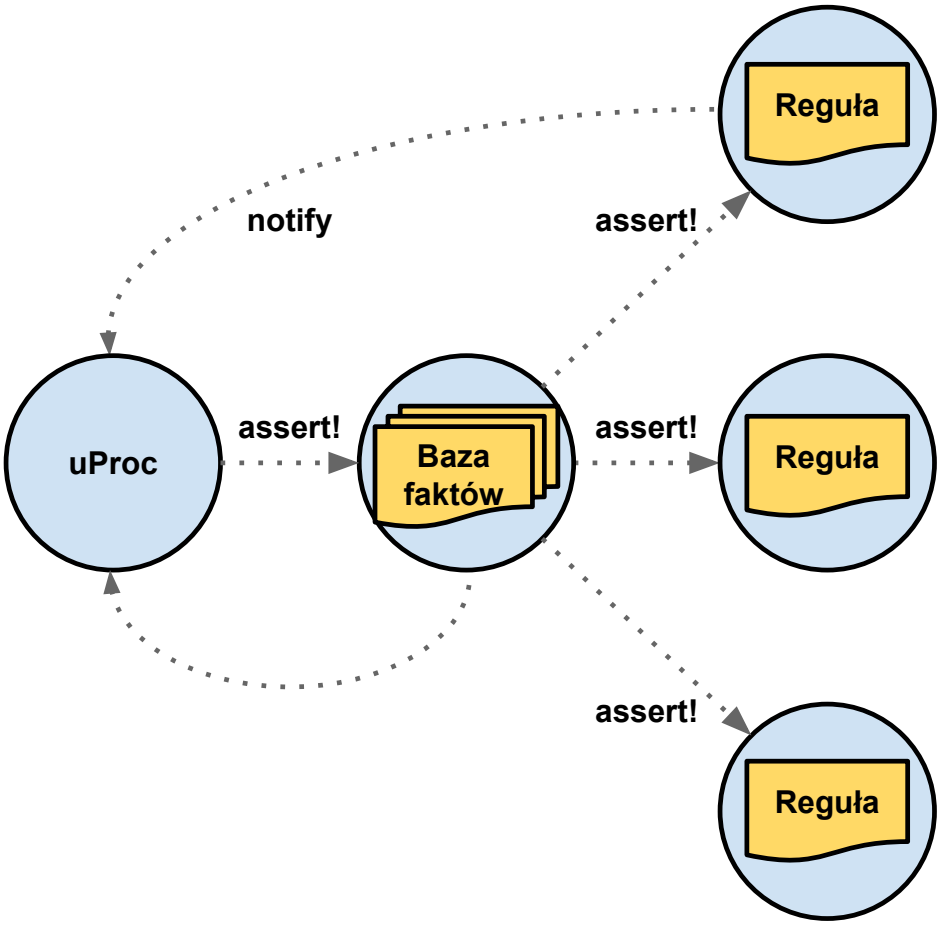
– querying fact store = create a rule and apply all known facts to it

5.5. Integracja z Systemem Uruchomieniowym

– describe how it sucks right now (notify-whenever instead of generic whenever, logic rule removal)

– describe possible integration with the module system (fact inference)

– describe possible representation of rules by autonomus processes [[47]]



Rysunek 5.3: Schemat działania rozproszonej wersji algorytmu *Rete*.

– hint at movig the implementation to the stdlib

6. Podsumowanie

– reiterate the goal of the thesis

– state how well has it been achieved

6.1. Kompilator języka FOOF

– needs better optimizations

– needs better error handling

6.2. Środowisko uruchomieniowe

– needs more stuff

– needs macroexpansion

– needs to drop RBS and move it into stdlib

6.3. Przyszłe kierunki rozwoju

– more datatypes

– native compilation via LLVM

– bootstrapping compiler

– librarized RBS

– librarized distribution with data encryption & ACLs

– data-level paralellism

Bibliografia

[1] P. E. McKenney, “Is parallel programming hard, and, if so, what can you do about it?.” Free online version.

[2] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[3] J. Backus, “Can programming be liberated from the von neumann style?: A functional style and its algebra of programs,” *Commun. ACM*, vol. 21, pp. 613–641, Aug. 1978.

[4] J. Höller, V. Tsiatsis, C. Mulligan, S. Karnouskos, S. Avesand, and D. Boyle, *From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence*. Elsevier, Apr. 2014.

[5] M. Richards, *Software Architecture Patterns*. O’Reilly, Feb. 2015.

[6] C. A. R. Hoare, “Hints on programming language design.,” tech. rep., Stanford, CA, USA, 1973.

[7] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews, *Revised [6] Report on the Algorithmic Language Scheme*. New York, NY, USA: Cambridge University Press, 1st ed., 2010.

[8] R. Milner, M. Tofte, and D. Macqueen, *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997.

[9] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, part I,” *Commun. ACM*, vol. 3, pp. 184–195, Apr. 1960.

[10] A. Church, “A set of postulates for the foundation of logic part I,” *Annals of Mathematics*, vol. 33, no. 2, pp. 346–366, 1932. http://www.jstor.org/stable/1968702Electronic Edition.

[11] A. Church, “A set of postulates for the foundation of logic part II,” *Annals of Mathematics*, vol. 34, no. 2, pp. 839–864, 1933.

[12] S. P. Jones and D. Lester, *Implementing functional languages: a tutorial*. Prentice Hall, 1992. Free online version.

[13] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge, MA, USA: MIT Press, 2nd ed., 1996.

[14] M. Felleisen and D. P. Friedman, "(Y Y) works!," 1991.

[15] K. Rzepecki, "Design of a programming language with support for distributed computing on heterogenous platforms." 2015.

[16] J. C. Reynolds, "The discoveries of continuations," *Lisp Symb. Comput.*, vol. 6, pp. 233–248, Nov. 1993.

[17] A. W. Appel, *Compiling with Continuations*. Cambridge University Press, 1992.

[18] R. Harper, "Programming in standard ml," 1998.

[19] R. K. Dybvig, S. P. Jones, and A. Sabry, "A monadic framework for delimited continuations," tech. rep., IN PROC, 2005.

[20] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, (San Francisco, CA, USA), pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[21] W. D. Clinger, "Foundations of actor semantics," tech. rep., Cambridge, MA, USA, 1981.

[22] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG (2Nd Ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[23] A. Bawden, "Quasiquotation in lisp," tech. rep., University of Aarhus, 1999.

[24] C. Queinnec, "Macroexpansion reflective tower," in *Proceedings of the Reflection'96 Conference*, pp. 93–104, 1996.

[25] J. N. Shutt, *Fexprs as the basis of Lisp function application or $vau: the ultimate abstraction*. PhD thesis, Worcester Polytechnic Institute, August 2010.

[26] A. Rossberg, "1ML - core and modules united," 2015.

[27] J. M. Gasbichler, *Fully-parameterized, first-class modules with hygienic macros*. PhD thesis, Eberhard Karls University of Tübingen, 2006. http://d-nb.info/980855152.

[28] D. S. C. G. Wang, W and K. Moessner, "Knowledge representation in the Internet of Things: Semantic modelling and its application," *Wang, W, De, S, Cassar, G and Moessner, K*, vol. 54, pp. 388 – 400.

[29] S. Hachem, T. Teixeira, and V. Issarny, "Ontologies for the Internet of Things," in *Proceedings of the 8th Middleware Doctoral Symposium*, MDS '11, (New York, NY, USA), pp. 3:1–3:6, ACM, 2011.

[30] H. Samimi, C. Deaton, Y. Ohshima, A. Warth, and T. Millstein, "Call by meaning," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, (New York, NY, USA), pp. 11–28, ACM, 2014.

[31] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[32] A. Ghuloum, "An Incremental Approach to Compiler Construction," in *Scheme and Functional Programming 2006*.

[33] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, (New York, NY, USA), pp. 111–122, ACM, 2004.

[34] G. Hutton and E. Meijer, "Monadic parser combinators," 1996.

[35] A. Bawden, "First-class macros have types," in *In 27th ACM Symposium on Principles of Programming Languages (POPL'00*, pp. 133–141, ACM, 2000.

[36] D.-A. German, "Recursion without circularity or using let to define letrec," 1995.

[37] O. Kaser, C. R. Ramakrishnan, and S. Pawagi, "On the conversion of indirect to direct recursion," vol. 2, pp. 151–164, Mar. 1993.

[38] D. Bacon, "A Hacker's Introduction to Partial Evaluation," 2002.

[39] D. Gudeman, "Representing type information in dynamically typed languages," 1993.

[40] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," *ACM Trans. Program. Lang. Syst.*, vol. 31, pp. 6:1–6:31, Feb. 2009.

[41] C. S. Pabla, "Completely fair scheduler," *Linux J.*, vol. 2009, Aug. 2009.

[42] R. Sedgewick, "Left-leaning red-black trees," 2008.

[43] P. Barnaghi, W. Wang, C. Henson, and K. Taylor, "Semantics for the Internet of Things: Early progress and back to the future," *Int. J. Semant. Web Inf. Syst.*, vol. 8, pp. 1–21, Jan. 2012.

[44] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19, no. 1, pp. 17 – 37, 1982.

[45] C. L. Forgy, *On the Efficient Implementation of Production Systems*. PhD thesis, Pittsburgh, PA, USA, 1979. AAI7919143.

[46] D. P. Miranker, *TREAT: A New and Efficient Match Algorithm for AI Production Systems*. PhD thesis, New York, NY, USA, 1987. UMI Order No. GAX87-10209.

[47] A. Gupta, C. Forgy, A. Newell, and R. Wedig, "Parallel algorithms and architectures for rule-based systems," *SIGARCH Comput. Archit. News*, vol. 14, pp. 28–37, May 1986.

A. Gramatyka języka FOOF

– concrete language grammar in PEG or BNF

B. Przykładowe programy

Poniżej zaprezentowano przykładowe programy w języku `FOOF` i krótki opis ich działania. Programy mogą zostać skompilowane i uruchomione za pomocą udostępnionego interfejsu kompilatora i środowiska uruchomieniowego języka. W konsoli systemu należy w tym celu wywołać odpowiednio funkcje `compile` i `run` podając interesujący program jako parametr, na przykład:

```
> (compile 'program)
> (run 'program)
```

B.1. Hello world!

Program definuje funkcję `hello` obrazującą podstawowe operacje języka i następnie wywołuje ją z jednym parametrem. Po uruchomieniu program powoduje wypisanie wiadomości `Hello world!` na ekranie komputera.

```
(define (hello world)
  (if (= nil world)
      (raise 'nope)
      (do (display "Hello ")
          (display world)
          (display "!")
          (newline))))

(hello "world")
```

Listing 21: Popularny program *Hello world!*.

B.2. Funkcja Fibonacciego

Program prezentuje definicję funkcji Fibonacciego z wykorzystaniem konstrukcji `letrec`, służącej do definiowania funkcji rekursywnych. Następnie program oblicza wynik funkcji Fibonacciego dla liczby 23.

```
(letrec ((fib (lambda (n)
                (if (< n 2)
                    n
                    (+ (fib (- n 1))
                       (fib (- n 2)))))))
  (fib 23))
```

Listing 22: Definicja funkcji Fibonacciego.

B.3. Obsługa błędów

Program prezentuje wykorzystanie wbudowanego w język systemu obsługi błędów. Deklarowana jest procedura obsługi błędów, która restartuje obliczenia z nową wartością. Następnie program dwukrotnie sygnalizuje wystąpienie błędu. Wynikiem działania programu jest liczba 24.

```
(* 2 (handle (raise (raise 3))
           (lambda (e restart)
             (restart (* 2 e)))))
```

Listing 23: Zastosowanie wbudowanego mechanizmu obsługi błędów.

B.4. Model Aktorowy

Program korzysta z dwóch komunikujących się procesów do zobrazowania sposobu wykorzystania zaimplementowanego w języku Modelu Aktorowego. Efektem działania programu jest wypisanie wiadomości `Hello world!` na ekranie komputera.

```
(let ((pid (spawn (lambda ()
                    (let ((msg (recv)))
                      (display (cdr msg))
                      (newline)
                      (send (car msg) " world!"))))))
  (send pid (cons (self) "Hello"))
  (display (recv))
  (newline))
```

Listing 24: Wykorzystanie prymitywnych operacji Modelu Aktorowego.

B.5. Współbieżne obliczenia funkcji Fibonacciego

Program definiuje funkcję Fibonacciego oraz dodatkową funkcję wyświetlającą informacje o systemie. Następnie tworzone są trzy procesy współbieżnie obliczające wartość funkcji Fibonacciego dla liczby 30. Program periodycznie wyświetla różne informacje o działających procesach.

```
(letrec ((fib (lambda (n)
                (if (< n 2)
                    n
                    (+ (fib (- n 1))
                       (fib (- n 2))))))
         (monitor (lambda ()
                    (task-info)
                    (sleep 2000)
                    (monitor))))
  (spawn (lambda ()
           (fib 30)))
  (spawn (lambda ()
           (fib 30)))
  (spawn (lambda ()
           (fib 30)))
  (monitor))
```

Listing 25: Równoległe obliczanie funkcji Fibonacciego.

B.6. System modułowy

Program definiuje dwa moduły - `logger` oraz `test`. Moduł `test` wymaga do działania implementacji modułu logowania. Program tworzy instancję modułu `logger` i następnie tworzy instancję modułu `test` wykorzystując uprzednio zdefiniowany moduł logowania. Efektem działania programu jest wypisanie dwóch wiadomości na ekranie komputera. Wiadomości są odpowiednio sformatowane przez moduł `logger`.

```
(module (logger)
  (define (log level string)
    (display "[")
    (display level)
    (display "] ")
    (display string)
    (newline))

  (define (debug string)
    (log 'DEBUG string))

  (define (info string)
    (log 'INFO string))

  (define (warn string)
    (log 'WARN string))

  (define (error string)
    (log 'ERROR string)))

(module (test logger)
  (define (do-something)
    (logger.info "doing something")
    (logger.error "failed badly!")))

(let ((t (test (logger))))
  (t.do-something))
```

Listing 26: Wykorzystanie wbudowanego systemu modułowego.

B.7. Wnioskowanie w przód

Program prezentuje wykorzystanie wbudowanego w język systemu regułowego. Definiowane są trzy funkcje, jedna z nich co pewien czas sygnalizuje zajście pewnego zdarzenia - upływ czasu. Druga funkcja oczekuje notyfikacji od systemu regułowego i wyświetla informacje o przechwyconych zdarzeniach. Trzecia funkcja, jest pomocniczą funkcją wyświetlającą informacje o procesach uruchomionych w systemie. Następnie program definiuje prostą regułę i uruchamia wszystkie niezbędne procesy.

```
(letrec ((monitor (lambda ()
                    (task-info)
                    (sleep 10000)
                    (monitor)))
         (timer (lambda (t)
                  (signal! `(curr-time ,t))
                  (sleep 1000)
                  (timer (+ t 1))))
         (listen (lambda ()
                   (let ((t (recv)))
                     (display "Current time: ")
                     (display (cdr (car t)))
                     (newline)
                     (listen)))))
  (spawn (lambda () (timer 0)))
  (notify-whenever (spawn (lambda ()
                            (listen)))
                   '(curr-time ?t))
  (monitor))
```

Listing 27: Wykorzystanie wbudowanego systemu regułowego.

B.8. Obsługa złożonych zdarzeń

Program działa podobnie do przykładu z listingu 27. Definiowana jest złożona reguła, która notyfikuje proces nasłuchujący jedynie, gdy wartości powiązane z faktami `foo` oraz `bar` osiągają odpowiednie wartości.

```
(letrec ((monitor (lambda ()
                    (task-info)
                    (sleep 10000)
                    (monitor)))
         (notify (lambda (prefix t)
                   (assert! `(notify ,prefix ,(random)))
                   (sleep t)
                   (notify prefix t)))
         (listen (lambda ()
                   (let ((m (recv)))
                     (display "Complex event: ")
                     (display m)
                     (newline)
                     (listen)))))
  (notify-whenever (spawn listen)
                   '(filter (and (?notify foo ?foo)
                                 (?notify bar ?bar))
                            (>= ?foo 0.5)
                            (< ?foo 0.75)
                            (<= ?bar 0.1)))
  (spawn (lambda ()
           (notify 'foo 1000)))
  (spawn (lambda ()
           (notify 'bar 5000)))
  (monitor))
```

Listing 28: Zastosowanie wbudowanego systemu regułowego do obsługi złożonych zdarzeń.

B.9. Wnioskowanie wstecz

Program prezentuje wykorzystanie wnioskowania wstecz wbudowanego w język systemu regułowego. Na bazie faktów wykonywany jest szereg operacji, a następnie program odpytuje

bazę faktów o wartości, dla których wystąpiły fakty `foo` oraz `bar`. Wynikiem działania programu jest asocjacja `(?value . 2)`.

```
(assert! '(foo 1))
(assert! '(foo 2))
(assert! '(foo 3))
(assert! '(bar 2))
(assert! '(bar 3))
(retract! '(foo 2))
(signal! '(foo 4))

(select '(and (foo ?value)
              (bar ?value)))
```

Listing 29: Wykorzystanie wnioskowania wstecz.

C. Spis wbudowanych funkcji i makr języka FOOF

- list contents of bootstrap.scm
- describe what `&make-structure`, `&yield-cont` etc do
- list available macros

D. Spisy rysunków i fragmentów kodu

Spis rysunków

1.1. Schemat interakcji poszczególnych elementów języka. 7

1.2. Przykład systemu opartego o heterogeniczną platformę sprzętową. 9

1.3. Przykład systemu heterogenicznego niezależnie od platformy sprzętowej. 10

3.1. Schemat poszczególnych faz kompilacji i przykładowych danych będących wynikiem ich działania. 24

4.1. Schemat architektury środowiska uruchomieniowego języka `FOOF`. 34

4.2. Schemat przykładowej reprezentacji typów danych języków funkcyjnych. 35

4.3. Schemat kontekstu procesu obrazujący rejestry niezbędne do jego działania. 39

4.4. Dodatkowe rejestry kontekstu mikroprocesu wymagade do implementacji algorytmu *Completely Fair Scheduler*. 40

4.5. Dodatkowe rejestry kontekstu mikroprocesu wymagade do implementacji Modelu Aktorowego. 41

4.6. Diagram obrazujący efekty przekazywania wiadomości pomiędzy mikroprocesami. 41

5.1. Schemat działania wbudowanych baz faktów i reguł. 43

5.2. Schemat łączenia podsieci w algorytmie *Rete*. 44

5.3. Schemat działania rozproszonej wersji algorytmu *Rete*. 45

Spis listingów

2.1. Podstawowe typy danych dostępne w języku FOOF. 12
2.2. Przykład implementacji wartości i operatorów logicznych w Rachunku Lambda. . . 13
2.3. Przykład ilustrujący działanie domknięć leksykalnych. 13
2.4. Przykład wykorzystania kontynuacji w języku FOOF. 15
2.5. Przykład wykorzystania mechanizmu obsługi błędów. 16
2.6. Przykład wykorzystania prymitywnych operacji Modelu Aktorowego w języku. . . 16
2.7. Przykład działania systemu makr w języku FOOF. 17
2.8. Przykład ilustrujący problem higieniczności systemu makr w języku Scheme. . . . 18
2.9. Przykład wykorzystania systemu modułowego języka FOOF. 19
2.10. Przykład wykorzystania prymitywnych operacji bazy wiedzy w języku. 21
3.11. Obsługa rozszerzeń składniowych S-wyrażeń w języku FOOF. 25
3.12. Pseudokod algorytmu makroekspansji w notacji języka FOOF. 26
3.13. Przykład działania algorytmu makroekspansji. 27
3.14. Przykład ekspansji makra module. 28
3.15. Przykład konwersji *Continuation Passing Style*. 29
3.16. Przykład transformacji konstrukcji letrec. 30
3.17. Przykład ilustrujący różnice pomiędzy algorytmami lambda-unoszenia oraz konwersji domknięć leksykalnych. 31
4.18. Porównanie wbudowanych typów danych języka FOOF i dialektu języka Scheme o nazwie Racket. 36
4.19. Przykład uruchomienia funkcji z listing 15. 37
4.20. Wykorzystanie kontynuacji do implementacji obsługi błędów. 38
B.21. Popularny program *Hello world!*. 55
B.22. Definicja funkcji Fibonacciego. 56
B.23. Zastosowanie wbudowanego mechanizmu obsługi błędów. 56
B.24. Wykorzystanie prymitywnych operacji Modelu Aktorowego. 57
B.25. Równoległe obliczanie funkcji Fibonacciego. 57
B.26. Wykorzystanie wbudowanego systemu modułowego. 58
B.27. Wykorzystanie wbudowanego systemu regułowego. 59

B.28. Zastosowanie wbudowanego systemu regułowego do obsługi złożonych zdarzeń. . 60
B.29. Wykorzystanie wnioskowania wstecz. 61