



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:	<i>Kajetan Rzepecki</i>
Kierunek studiów:	<i>Informatyka</i>
Opiekun pracy:	<i>dr inż. Piotr Matyasik</i>

Kraków, 2015

*Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nie-
prawdy, 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	19
2.8. Inżynieria wiedzy w języku	20
3. Kompilator języka FOOF	23
3.1. Architektura kompilatora	23
3.2. Parser	23
3.3. Makro-ekspansja	24
3.4. Obsługa Systemu Modułowego	24
3.5. Transformacja <i>Continuation Passing Style</i>	24
3.6. Generacja kodu	25
4. Środowisko uruchomieniowe języka	27
4.1. Architektura środowiska uruchomieniowego	27
4.2. Implementacja podstawowych typów danych	28
4.3. Implementacja kontynuacji	28
4.4. Implementacja obsługi wyjątków	28
4.5. Implementacja procesów	28
4.6. Harmonogramowanie procesów	30
4.7. Implementacja Modelu Aktorowego	30

4.8. Dystrybucja obliczeń	32
5. Reprezentacja i przetwarzanie wiedzy	33
5.1. Reprezentacja wiedzy w języku	33
5.2. Algorytm Rete	33
5.3. Implementacja Rete - wnioskowanie w przód	34
5.4. Implementacja wnioskowania wstecz	34
5.5. Integracja z Systemem Uruchomieniowym	34
6. Podsumowanie	37
6.1. Kompilator języka FOOF	37
6.2. Środowisko uruchomieniowe	37
6.3. Przyszłe kierunki rozwoju	37
Bibliografia	39
A. Gramatyka języka FOOF	43
B. Przykładowe programy	45
B.1. Hello world!	45
B.2. Funkcja Fibonacciego	46
B.3. Obsługa błędów	46
B.4. Model Aktorowy	46
B.5. Współbieżne obliczenia funkcji Fibonacciego	47
B.6. System modułowy	48
B.7. Wnioskowanie w przód	49
B.8. Obsługa złożonych zdarzeń	50
B.9. Wnioskowanie wstecz	50
C. Spis wbudowanych funkcji języka FOOF	53
D. Spisy rysunków i fragmentów kodu	55

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 następnej sekcji.

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.

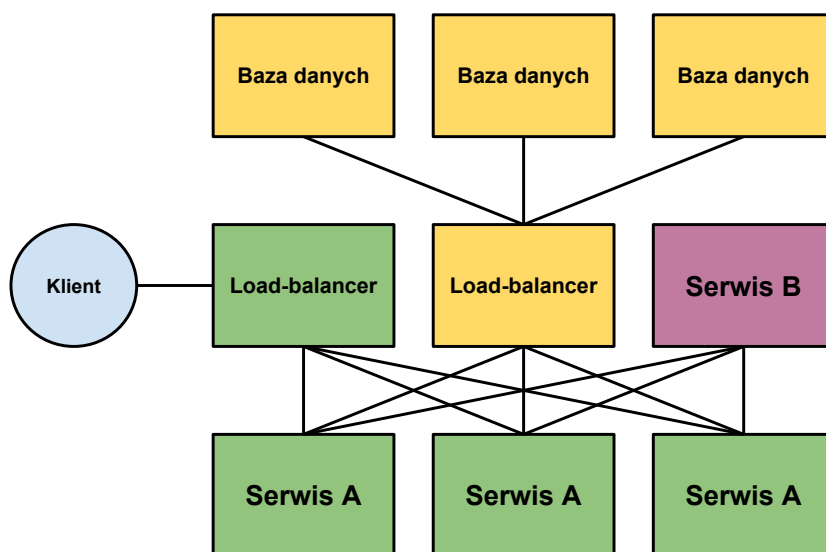


Rysunek 1.2: Przykład systemu opartego o heterogeniczną platformę sprzętową.

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

Problem heterogeniczności dotyczy 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 mikroservisową [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 pierwszym przypadku heterogeniczność wynika ze zróżnicowania maszyn należących do platformy sprzętowej, natomiast w drugim wynika ona z istnienia mikroservisów, które realizują pojedyncze, konkretnie 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 FOOF¹, ma być uosobieniem ideologii świadomości platformy.

1.2. Zawartość pracy

- list what is found where in the thesis

¹Nazwa 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 poczynając 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 literalową.

```
1 23.5
2 symbol
3 :symbol
4 "ciąg znaków"
5 (1 2 3)
6 [1 2 3]
7 {: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 literalowej 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 osadzone 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 conceptualnie 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.

```

1  TRUE :=  $\lambda x.\lambda y.x$ 
2  FALSE :=  $\lambda x.\lambda y.y$ 
3
4  AND :=  $\lambda p.\lambda q.p\ q\ p$ 
5  OR :=  $\lambda p.\lambda q.p\ p\ q$ 
6  NOT :=  $\lambda p.\lambda a.\lambda b.p\ b\ a$ 
7
8  AND TRUE FALSE
9       $\equiv (\lambda p.\lambda q.p\ q\ p)\ TRUE\ FALSE \rightarrow_{\beta} TRUE\ FALSE\ TRUE$ 
10      $\equiv (\lambda x.\lambda y.x)\ FALSE\ TRUE \rightarrow_{\beta} FALSE$ 

```

Listing 2: Przykład implementacji wartości i operatorów logicznych jedynie za pomocą wyrażeń 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.

```
1 (let* ((x 23)
2       (foo (lambda () x)))
3 (let ((x 5))
4   (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 funkcji. 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 F00F 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 zdecydować 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 związku 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 `shift` oraz `reset` opisane w [19], w celu zapewnienia większej kontroli nad przepływem sterowania programem. 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 konstrukcjom `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 konstrukcji `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.

```
1 (handle (do ...  
2         (raise 'error) ;; Błąd w trakcie wykonywania obliczeń.  
3         ...)  
4 (lambda (error restart)  
5     ...  
6     (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 wykorzystując 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 przysyłanie 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.


```
1 (send (spawn (lambda ()
2             (sleep 1000)
3             (send (recv) 'message)))
4     (self))
5
6 (equal? (recv) 'message)
```

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

Program ten tworzy nowy proces korzystając z funkcji `spawn`, któremu natychmiastowo wysła wiadomość za pośrednictwem funkcji `send`, zawierając w niej swój identyfikator `self`, po czym przechodzi do oczekiwania na odpowiedź wywołując funkcję `recv`. Tymczasem, nowo powstały proces zostaje uspiony 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

Prawdopodobnie 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.

<pre> 1 ;; Przed makroekspansją: 2 (and 23 42) 3 4 5 6 (let ((x 23)) 7 (display x)) 8 9 10 `(4 is ,(* 2 2)) </pre>	<pre> 1 ;; Po makroekspansji: 2 (if 23 3 42 4 false) 5 6 ((lambda (x) 7 (display x)) 8 23) 9 10 (list '4 'is (* 2 2)) </pre>
--	---

Listing 7: Przykład działania systemu makr w języku FOOF.

Efekt 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 często borykają się z problemami **higieniczności** generowanego kodu. Problem ten ilustruje przykład z listingu 8.

```

1 (define-macro (unless c . b)
2   `(if (not ,c)
3       (begin ,@b)
4       #void))
5
6 (let ((not identity))
7   (unless #t
8     (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 komponenty - 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** dostarczająca 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 następnej 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 do-

kładności pomiarów niż dotychczasowo dostępna, powinien bez modyfikacji programu, ani ingerencji jego użytkownika, zacząć z niego korzystać. Natomiast, w przypadku katastrofalnego błędu rzeczonoego 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 sobą pewnymi zależnościami. Ontologia ta może być następnie odpytwana, 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 zastosowanym 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.

```
1 (whenever set-of-conditions
2   (lambda (_)
3     (retract! some-fact)
4     (assert! another-fact)))
5 (signal! an-event)
```

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

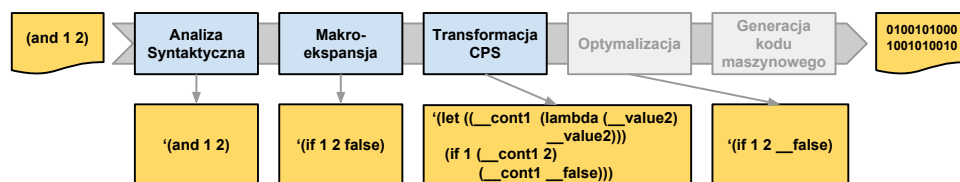
Przykład ten definiuje jedną regułę bez zagłębiania się w szczegóły implementacji systemu regułowego języka FOOF. Reguła ta, bazując na spełnialności pewnego zbioru warunków **set-of-conditions** modyfikuje wbudowaną bazę faktów przez **asercję** i **retrakcję** różnych faktów. Dodatkowo, przykład **sygnalizuje** 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 poprzedniej sekcji problemu linkowania modułów. Rozwiązanie to polega na przetwarzaniu 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 protokoły, które są konieczne i wystarczające do poprawnego, wspólnego działania, a system regułowy automatycznie wybierze spełniające je, dostępne submoduły.

3. Kompilator języka FOOF

- mention technology selection & limitations (large project, little time) [[31]]
- mention possible bootstrapping
- briefly touch on the architecture [[32]]
- hint at using Scheme for the boring details (datatypes etc)

3.1. Architektura kompilatora



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

- list compilation phases [[32]] [[31]] [[17]]
- list which phases have been actually implemented
- list which phases have been skipped and say why (optimization, code-gen, parsing)

3.2. Parser

- briefly describe how Scheme praser works and what it produces [[7]] [[13]]
- hint at a possibility of replacing this with a PEG-based packrat [[33]] [[34]]
- note about special quasiquote syntax [[23]]

3.3. Makro-ekspansja

- describe macroexpansion phase
- describe why macroexpansion is hardcoded [[27]]
- list available macros
- show some examples of macro-expanded code

3.4. Obsługa Systemu Modułowego

- describe how modules are handled right now [[27]] [[26]]
- show some examples of macro-expanded structures & modules
- maby combine this with the previous section ?
- maby hint at special module access syntax (foo.bar.baz)

3.5. Transformacja *Continuation Passing Style*

- reformat this to fit the narrative

Do tego celu służy technika przekazywania kontynuacji (ang. *Continuation Passing Style*, *CPS*) polegająca na automatycznej transformacji kodu źródłowego programu do formatu, w którym wszystkie funkcje przyjmują dodatkowy argument będący kontynuacją, którą przekazują dalej [17].

Listing 11 prezentuje przykład konwersji CPS prostej funkcji.

<pre>1 ;; Styl aplikatywny: 2 (lambda (x y) 3 (* 2 (+ x y)))</pre>	<pre>1 ;; Styl Continuation Passing: 2 (lambda (x y cont) 3 (+& x y 4 (lambda (v) 5 (*& 2 v cont)))) 6 7 ;; Transformacja wbudowanych funkcji: 8 (define (+& a b cont) 9 (cont (+ a b)))</pre>
--	--

Listing 11: 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.

- describe in detail how to transform simple stuff
- describe in detail how to transform functions (recursion problems & crude solution via mutation [[35]], [[36]], [[14]])
- describe in detail how to handle exceptions
- describe in detail why this is useful (partial evaluation, constant folding etc) [[37]]
- hint at emitting calls to primitive functions `&yield-cont`, `&uproc-error-handler` etc

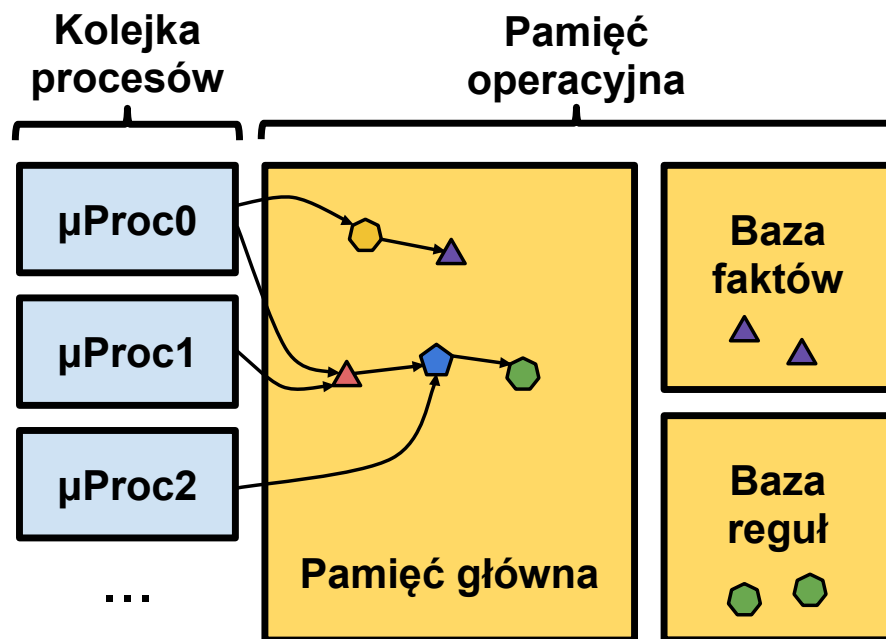
3.6. Generacja kodu

- describe how a subset of both Scheme and FOOF is emitted (contrast with Core Erlang) [[38]] [[39]]
- describe how Scheme is used for direct code execution
- hint at further development using LLVM [[?]]
- mention a requirement to perform closure conversion or lambda lifting [[12]]
- add a code example contrasting closure conversion and lambda lifting

4. Środowisko uruchomieniowe języka

- briefly touch on the architecture
- mention Scheme bootstrap

4.1. Architektura środowiska uruchomieniowego



Rysunek 4.1: Schemat architektury środowiska uruchomieniowego języka FOF.

- describe various parts
- mention that this is single threaded and requires forking for real concurrency
- hint at in-depth description of RBS implementation in a future section

4.2. Implementacja podstawowych typów danych

- describe scheme bootstrap [[7]]
- describe equivalence of various constructs such as lambdas

4.3. Implementacja kontynuacji

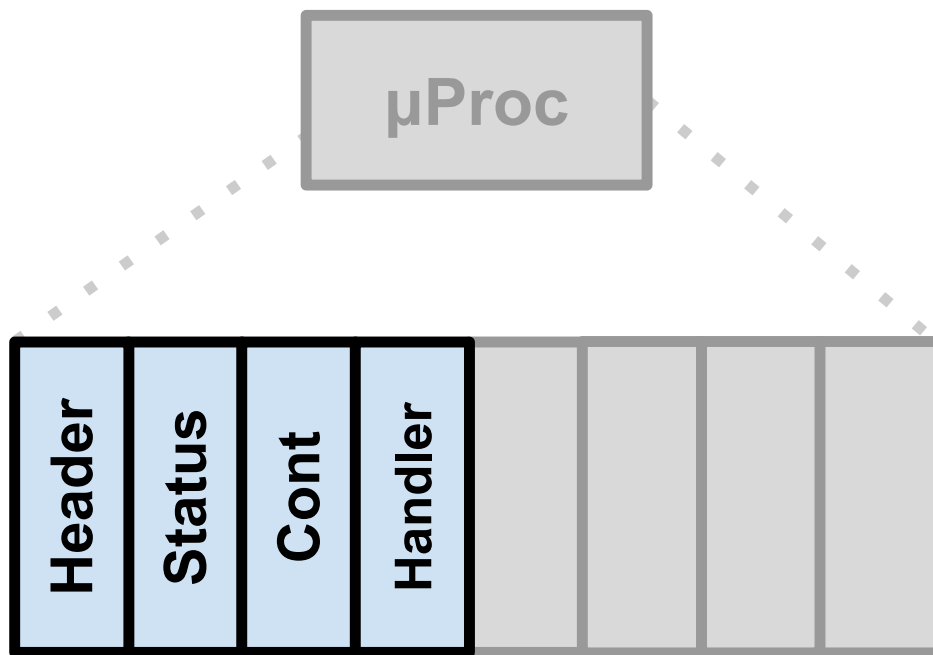
- describe how continuations are handled without getting into CFS (returning cont + hole aka trampoline, contrast to how G-machine/TIM reductions work) [[17]] [[12]]
- add a code example with step-by-step execution
- hint at debugging potential using step by step continuation execution with debug info inbetween

4.4. Implementacja obsługi wyjątków

- describe how continuations are used for error handling - handle & raise
- note about restarts
- note about implementing letcc using handle & raise ?

4.5. Implementacja procesów

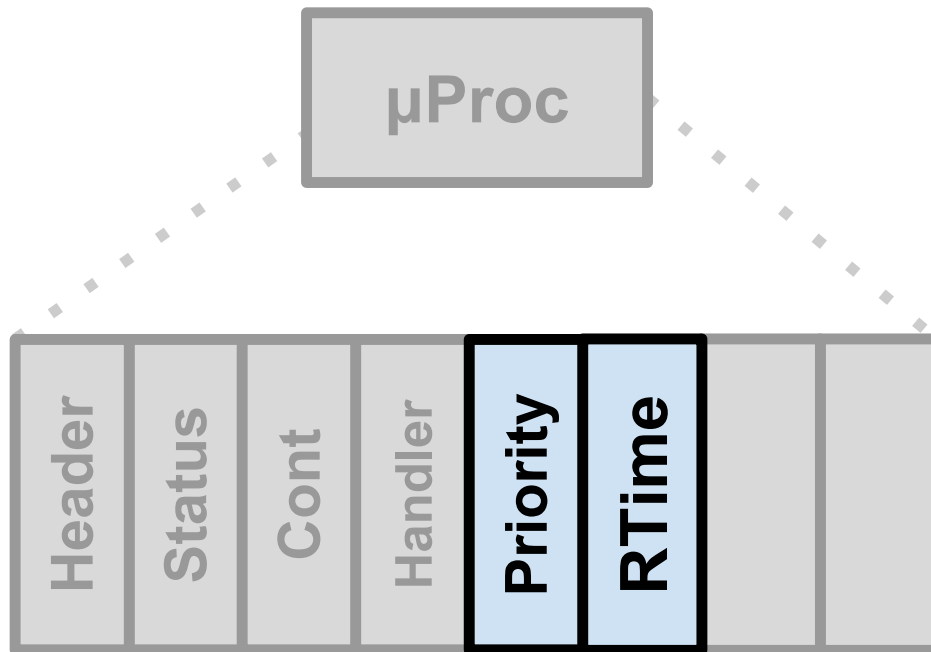
- add a diagram of the uProc context - only include status, cont & handler registers



Rysunek 4.2: 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

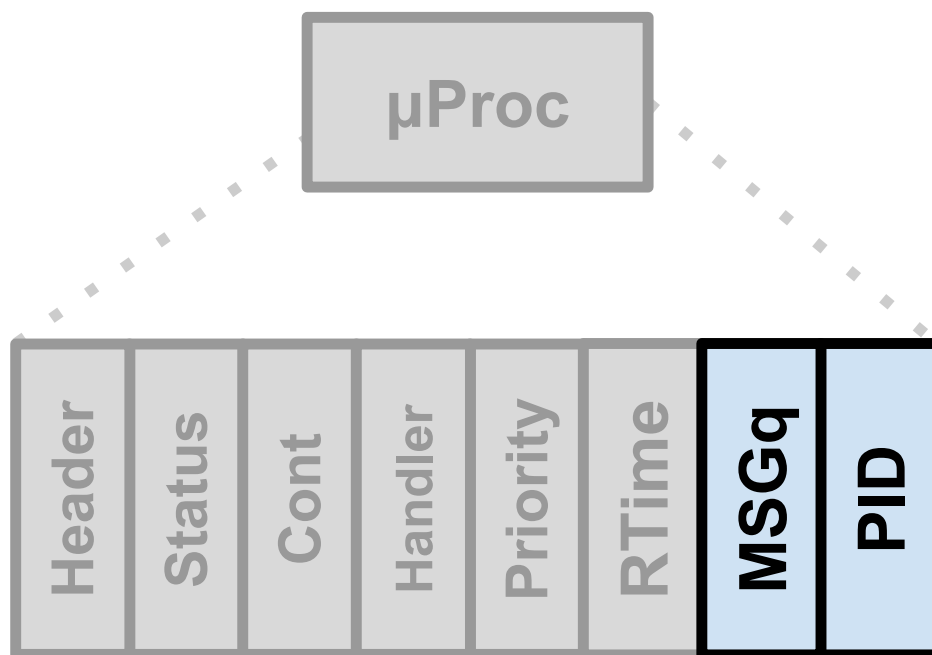


Rysunek 4.3: Dodatkowe rejestry kontekstu mikroprocesu wymagane do implementacji algorytmu *Completely Fair Scheduler*.

- describe the Completely Fair Scheduler [[41]]
- add pseudocode listing showing the algorithm
- describe μProc 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]]

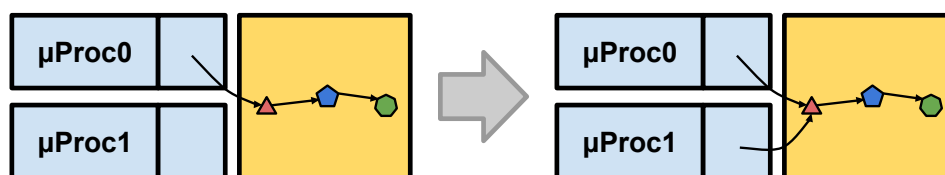
4.7. Implementacja Modelu Aktorowego

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



Rysunek 4.4: Dodatkowe rejestry kontekstu mikroprocesu wymagane 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 4.5: Diagram obrazujący efekty przekazywania wiadomości pomiędzy mikroprocesami.

- add some code examples and discussion of its effects and what happens
- contrast with erlang [[22]]

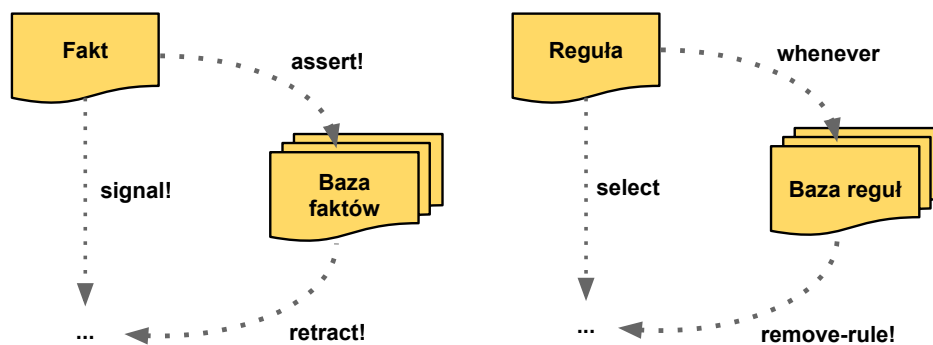
4.8. Dystrybucja obliczeń

- 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

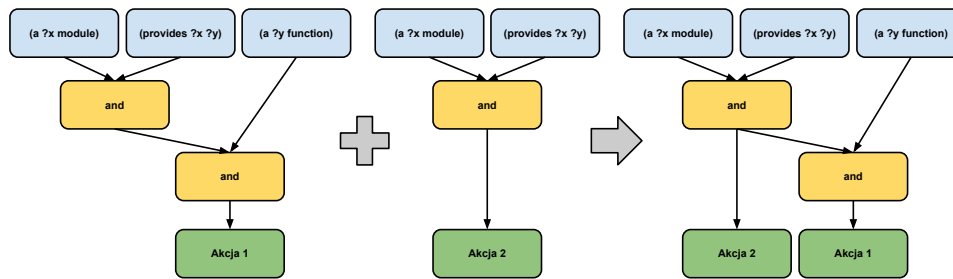


Rysunek 5.1: Schemat działania wbudowanych baz faktów i reguł.

- 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]]

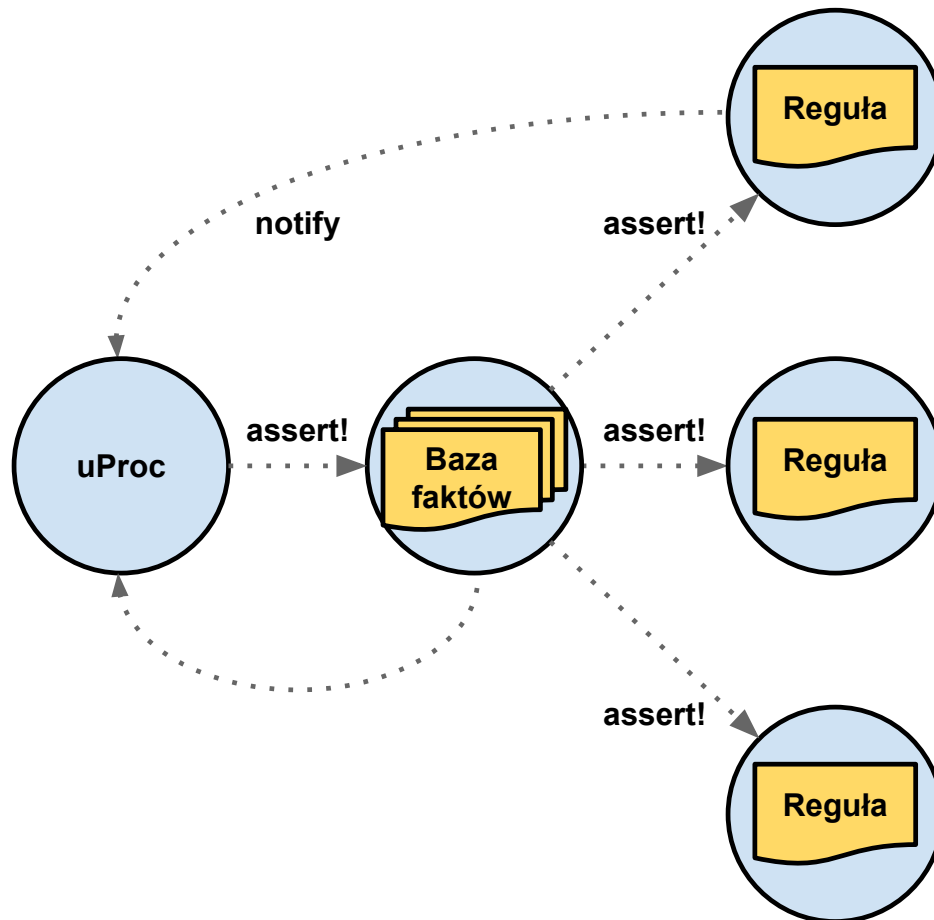
5.4. Implementacja wnioskowania wstecz

- 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 moving the implementation to the stdlib

6. Podsumowanie

- reiterate the goal of the thesis
- state how well has it been achieved

6.1. Kompilator języka F00F

- 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 parallelism

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/1968702>Electronic 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. Ghuloum, “An Incremental Approach to Compiler Construction,” in *Scheme and Functional Programming 2006*.
- [32] 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.
- [33] G. Hutton and E. Meijer, “Monadic parser combinators,” 1996.
- [34] 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.
- [35] D.-A. German, “Recursion without circularity or using let to define letrec,” 1995.
- [36] O. Kaser, C. R. Ramakrishnan, and S. Pawagi, “On the conversion of indirect to direct recursion,” vol. 2, pp. 151–164, Mar. 1993.
- [37] D. Bacon, “A Hacker’s Introduction to Partial Evaluation,” 2002.
- [38] R. Carlsson, “An introduction to Core Erlang,” in *In Proceedings of the PLI’01 Erlang Workshop*, 2001.
- [39] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding, “Core Erlang 1.0.3 language specification,” tech. rep., Department of Information Technology, Uppsala University, Nov. 2004.
- [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 F00F

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

```
1 (define (hello world)
2   (if (= nil world)
3       (raise 'nope)
4       (do (display "Hello ")
5           (display world)
6           (display "!")
7           (newline))))
8
9 (hello "world")
```

Listing 12: 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.

```
1 (letrec ((fib (lambda (n)
2             (if (< n 2)
3                 n
4                 (+ (fib (- n 1))
5                   (fib (- n 2)))))))
6 (fib 23))
```

Listing 13: 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.

```
1 (* 2 (handle (raise (raise 3))
2             (lambda (e restart)
3               (restart (* 2 e)))))
```

Listing 14: 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.

```

1 (let ((pid (spawn (lambda ()
2                     (let ((msg (recv)))
3                         (display (cdr msg))
4                         (newline)
5                         (send (car msg) " world!"))))))
6   (send pid (cons (self) "Hello"))
7   (display (recv))
8   (newline))

```

Listing 15: 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.

```

1 (letrec ((fib (lambda (n)
2                 (if (< n 2)
3                     n
4                     (+ (fib (- n 1))
5                         (fib (- n 2))))))
6   (monitor (lambda ()
7               (task-info)
8               (sleep 2000)
9               (monitor))))
10  (spawn (lambda ()
11           (fib 30)))
12  (spawn (lambda ()
13           (fib 30)))
14  (spawn (lambda ()
15           (fib 30)))
16  (monitor))

```

Listing 16: 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`.

```
1 (module (logger)
2   (define (log level string)
3     (display "[" )
4     (display level)
5     (display "]" )
6     (display string)
7     (newline))
8
9   (define (debug string)
10    (log 'DEBUG string))
11
12   (define (info string)
13    (log 'INFO string))
14
15   (define (warn string)
16    (log 'WARN string))
17
18   (define (error string)
19    (log 'ERROR string)))
20
21 (module (test logger)
22   (define (do-something)
23     (logger.info "doing something")
24     (logger.error "failed badly!")))
25
26 (let ((t (test (logger))))
27   (t.do-something))
```

Listing 17: 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.

```
1 (letrec ((monitor (lambda ()
2                     (task-info)
3                     (sleep 10000)
4                     (monitor)))
5   (timer (lambda (t)
6             (signal! `(curr-time ,t))
7             (sleep 1000)
8             (timer (+ t 1))))
9   (listen (lambda ()
10            (let ((t (recv)))
11              (display "Current time: ")
12              (display (cdr (car t)))
13              (newline)
14              (listen)))))
15 (spawn (lambda () (timer 0)))
16 (notify-whenever (spawn (lambda ()
17                           (listen)))
18                  '(curr-time ?t))
19 (monitor))
```

Listing 18: Wykorzystanie wbudowanego systemu regułowego.

B.8. Obsługa złożonych zdarzeń

Program działa podobnie do przykładu z listingu 18. Definiowana jest złożona reguła, która notyfikuje proces nasłuchujący jedynie, gdy wartości powiązane z faktami `foo` oraz `bar` osiągną odpowiednie wartości.

```
1 (letrec ((monitor (lambda ()
2     (task-info)
3     (sleep 10000)
4     (monitor)))
5   (notify (lambda (prefix t)
6     (assert! `(notify ,prefix ,(random)))
7     (sleep t)
8     (notify prefix t)))
9   (listen (lambda ()
10     (let ((m (recv)))
11       (display "Complex event: ")
12       (display m)
13       (newline)
14       (listen))))))
15 (notify-whenever (spawn listen)
16   '(filter (and (?notify foo ?foo)
17             (?notify bar ?bar))
18             (>= ?foo 0.5)
19             (< ?foo 0.75)
20             (<= ?bar 0.1)))
21 (spawn (lambda ()
22   (notify 'foo 1000)))
23 (spawn (lambda ()
24   (notify 'bar 5000)))
25 (monitor))
```

Listing 19: 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`).

```
1  (assert! '(foo 1))
2  (assert! '(foo 2))
3  (assert! '(foo 3))
4  (assert! '(bar 2))
5  (assert! '(bar 3))
6  (retract! '(foo 2))
7  (signal! '(foo 4))
8
9  (select '(and (foo ?value)
10           (bar ?value)))
```

Listing 20: Wykorzystanie wnioskowania wstecz.

C. Spis wbudowanych funkcji języka F00F

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

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.	23
4.1. Schemat architektury środowiska uruchomieniowego języka FOOF.	27
4.2. Schemat kontekstu procesu obrazujący rejestry niezbędne do jego działania.	29
4.3. Dodatkowe rejestry kontekstu mikroprocesu wymagane do implementacji algorytmu <i>Completely Fair Scheduler</i>	30
4.4. Dodatkowe rejestry kontekstu mikroprocesu wymagane do implementacji Modelu Aktorowego.	31
4.5. Diagram obrazujący efekty przekazywania wiadomości pomiędzy mikroprocesami.	31
5.1. Schemat działania wbudowanych baz faktów i reguł.	33
5.2. Schemat łączenia podsieci w algorytmie <i>Rete</i>	34
5.3. Schemat działania rozproszonej wersji algorytmu <i>Rete</i>	35

Spis listingów

2.1. Podstawowe typy danych dostępne w języku FOOF.	12
2.2. Przykład implementacji wartości i operatorów logicznych jedynie za pomocą wyrażeń lambda.	13
2.3. Przykład ilustrujący działanie domknięć leksykalnych.	14
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. . .	17
2.7. Przykład działania systemu makr w języku FOOF.	18
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. Przykład konwersji <i>Continuation Passing Style</i>	24
B.12. Popularny program <i>Hello world!</i>	45
B.13. Definicja funkcji Fibonacciego.	46
B.14. Zastosowanie wbudowanego mechanizmu obsługi błędów.	46
B.15. Wykorzystanie prymitywnych operacji Modelu Aktorowego.	47
B.16. Równoległe obliczanie funkcji Fibonacciego.	47
B.17. Wykorzystanie wbudowanego systemu modułowego.	48
B.18. Wykorzystanie wbudowanego systemu regułowego.	49
B.19. Zastosowanie wbudowanego systemu regułowego do obsługi złożonych zdarzeń. .	50
B.20. Wykorzystanie wnioskowania wstecz.	51