Wprowadzenie do jezyka LISP

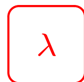
Lech Błażejewski, Piotr Kaźmierczak, PJWSTK

Ekspert programowania odwiedził Johna McCarthy'ego by dowiedzieć sie czegoś o jezyku LISP. McCarthy podał herbate. Napełnił filiżanke gościa, jednak nie przestał nalewać gdy ta zaczeła sie przepełniać. Ekspert w zdumieniu obserwował lejacy sie z dzbana płyn, wkońcu nie wytrzymał:

— *Filiżanka jest pełna. Nie zmieści sie już ani kropla!*

— *Podobnie jak ta filiżanka* – powiedział McCarthy – *jesteś pełen swoich własnych opinii i spekulacji. Jak zatem moge ukazać Ci czym jest LISP, dopóki nie opróżnisz filiżanki?*

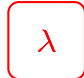
Polsko–Japońska Wyższa Szkoła Technik Komputerowych • Warszawa • 10 maja 2004



Podstawowa charakterystyka jezyka

✓ Wszystkie funkcje i programy sa wyrażeniami zwracajacymi wartość
⇒ *"LISP programmers know the value of everything and the cost of nothing"*.

✓ Niezwykła rzecza w LISP jest to, że może on być napisany sam w sobie ;-)
⇒ *"LISP is a programmable programming language"*.



Podstawowa charakterystyka jezyka

✓ Wszystkie funkcje i programy sa wyrażeniami zwracajacymi wartość
⇒ *"LISP programmers know the value of everything and the cost of nothing"*.

✓ Niezwykła rzecza w LISP jest to, że może on być napisany sam w sobie ;-)
⇒ *"LISP is a programmable programming language"*.

Historia LISP: narodziny jezyka

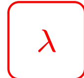
✓ Kluczowe koncepcje
⇒ John McCarthy (profesor EE z MIT) (1956 - *Darmouth Summer Research Project on AI*) oraz Marvin Minsky (profesor matematyki z MIT)
⇒ MIT Artificial Intelligence Project (1958).

✓ Motywacja
⇒ jezyk algebraiczny do przetwarzania list na potrzeby AI.

✓ Pierwsze dialekty na komputerach IBM 704, PDP-1/6/10.

✓ Podstawowy dialekt Lispa ⇒ Lisp 1.5 (1960-65).

✓ Dwa główne dialekty ⇒ MacLisp oraz InterLisp (koniec lat '70).



Historia LISP: narodziny jezyka

✓ Kluczowe koncepcje
⇒ John McCarthy (profesor EE z MIT) (1956 - *Darmouth Summer Research Project on AI*) oraz Marvin Minsky (profesor matematyki z MIT)
⇒ MIT Artificial Intelligence Project (1958).

✓ Motywacja
⇒ jezyk algebraiczny do przetwarzania list na potrzeby AI.

✓ Pierwsze dialekty na komputerach IBM 704, PDP-1/6/10.

✓ Podstawowy dialekt Lispa ⇒ Lisp 1.5 (1960-65).

✓ Dwa główne dialekty ⇒ MacLisp oraz InterLisp (koniec lat '70).

Historia LISP: narodziny jezyka

- ✓ Kluczowe koncepcje
  ⇒ John McCarthy (profesor EE z MIT) (1956 - *Darmouth Summer Research Project on AI*) oraz Marvin Minsky (profesor matematyki z MIT)
  ⇒ MIT Artificial Intelligence Project (1958).
- ✓ Motywacja
  ⇒ jezyk algebraiczny do przetwarzania list na potrzeby AI.
- ✓ Pierwsze dialekty na komputerach IBM 704, PDP-1/6/10.
- ✓ Podstawowy dialekt Lispa ⇒ Lisp 1.5 (1960-65).
- ✓ Dwa główne dialekty ⇒ MacLisp oraz InterLisp (koniec lat '70).

Historia LISP: narodziny jezyka

- ✓ Kluczowe koncepcje
  ⇒ John McCarthy (profesor EE z MIT) (1956 - *Darmouth Summer Research Project on AI*) oraz Marvin Minsky (profesor matematyki z MIT)
  ⇒ MIT Artificial Intelligence Project (1958).
- ✓ Motywacja
  ⇒ jezyk algebraiczny do przetwarzania list na potrzeby AI.
- ✓ Pierwsze dialekty na komputerach IBM 704, PDP-1/6/10.
- ✓ Podstawowy dialekt Lispa ⇒ Lisp 1.5 (1960-65).
- ✓ Dwa główne dialekty ⇒ MacLisp oraz InterLisp (koniec lat '70).

Historia LISP: narodziny jezyka

- ✓ Kluczowe koncepcje
  ⇒ John McCarthy (profesor EE z MIT) (1956 - *Darmouth Summer Research Project on AI*) oraz Marvin Minsky (profesor matematyki z MIT)
  ⇒ MIT Artificial Intelligence Project (1958).
- ✓ Motywacja
  ⇒ jezyk algebraiczny do przetwarzania list na potrzeby AI.
- ✓ Pierwsze dialekty na komputerach IBM 704, PDP-1/6/10.
- ✓ Podstawowy dialekt Lispa ⇒ Lisp 1.5 (1960-65).
- ✓ Dwa główne dialekty ⇒ MacLisp oraz InterLisp (koniec lat '70).

Historia LISP: zmiany miedzy wersja 1 a 1.5 jezyka

- ✓ Dodawanie elementów do listy oraz ich usuwanie.
- ✓ Rozróżnienie miedzy wyrażeniami a liczbami.
- ✓ Funkcje ze zmienna liczba argumentów (przekazywanie do funkcji listy argumentów jako parametr zamiast każdego argumentu osobno).

Historia LISP: zmiany miedzy wersja 1 a 1.5 jezyka

- ✓ Dodawanie elementów do listy oraz ich usuwanie.
- ✓ Rozróżnienie miedzy wyrażeniami a liczbami.
- ✓ Funkcje ze zmienna liczba argumentów (przekazywanie do funkcji listy argumentów jako parametr zamiast każdego argumentu osobno).

Historia LISP: zmiany miedzy wersja 1 a 1.5 jezyka

- ✓ Dodawanie elementów do listy oraz ich usuwanie.
- ✓ Rozróżnienie miedzy wyrażeniami a liczbami.
- ✓ Funkcje ze zmienna liczba argumentów (przekazywanie do funkcji listy argumentów jako parametr zamiast każdego argumentu osobno).

Historia LISP: dialekt MacLisp

✓ Ulepszona wersja Lisp 1.5.

✓ Ulepszona obsługa błedów.

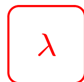
✓ Funkcje o zmiennej liczbie argumentów.

✓ Po raz pierwszy pojawiaja sie makra.

✓ Tablice.

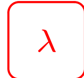
✓ Przyspieszenie operacji arytmetycznych.

✓ Porzadny kompilator jezyka.



Historia LISP: dialekt MacLisp

- ✓ Ulepszona wersja Lisp 1.5.
- ✓ Ulepszona obsługa błedów.
- ✓ Funkcje o zmiennej liczbie argumentów.
- ✓ Po raz pierwszy pojawiaja sie makra.
- ✓ Tablice.
- ✓ Przyspieszenie operacji arytmetycznych.
- ✓ Porzadny kompilator jezyka.



Historia LISP: dialekt MacLisp

- ✓ Ulepszona wersja Lisp 1.5.
- ✓ Ulepszona obsługa błedów.
- ✓ Funkcje o zmiennej liczbie argumentów.
- ✓ Po raz pierwszy pojawiaja sie makra.
- ✓ Tablice.
- ✓ Przyspieszenie operacji arytmetycznych.
- ✓ Porzadny kompilator jezyka.

Historia LISP: dialekt MacLisp

✓ Ulepszona wersja Lisp 1.5.

✓ Ulepszona obsługa błedów.

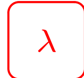
✓ Funkcje o zmiennej liczbie argumentów.

✓ Po raz pierwszy pojawiaja sie makra.

✓ Tablice.

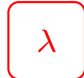
✓ Przyspieszenie operacji arytmetycznych.

✓ Porzadny kompilator jezyka.



Historia LISP: dialekt MacLisp

- ✓ Ulepszona wersja Lisp 1.5.
- ✓ Ulepszona obsługa błedów.
- ✓ Funkcje o zmiennej liczbie argumentów.
- ✓ Po raz pierwszy pojawiaja sie makra.
- ✓ Tablice.
- ✓ Przyspieszenie operacji arytmetycznych.
- ✓ Porzadny kompilator jezyka.



Historia LISP: dialekt MacLisp

✓ Ulepszona wersja Lisp 1.5.

✓ Ulepszona obsługa błedów.

✓ Funkcje o zmiennej liczbie argumentów.

✓ Po raz pierwszy pojawiaja sie makra.

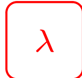
✓ Tablice.

✓ Przyspieszenie operacji arytmetycznych.

✓ Porzadny kompilator jezyka.

Historia LISP: dialekt MacLisp

- ✓ Ulepszona wersja Lisp 1.5.
- ✓ Ulepszona obsługa błedów.
- ✓ Funkcje o zmiennej liczbie argumentów.
- ✓ Po raz pierwszy pojawiaja sie makra.
- ✓ Tablice.
- ✓ Przyspieszenie operacji arytmetycznych.
- ✓ Porzadny kompilator jezyka.



Historia LISP: dialekt Interlisp

✓ Wprowadzono szereg innowacji do metodologii programowania.

✓ Konstrukt iteracyjny ⇒ makro `loop` używane na Lisp Machines oraz w dialekcie MacLisp, aktualnie obecne w Common Lisp.

(Czesto określane jako jedna z najwiekszych wad jezyka ponieważ narusza podstawowa zasade, mówiaca o tym, że symbol powinien wystepować tylko jako pierwszy element s-wyrażenia a wszystkie pozostałe wyrażenia powinny być przedstawione jako zagnieżdżona struktura listowa. Petla `loop` to jeden z nielicznych przypadków (jedyny?) w standardzie Common Lisp, gdzie mamy doczynienia ze zorientowaniem na słowo kluczowe (*keyword-oriented*)).

Historia LISP: dialekt Interlisp

- ✓ Wprowadzono szereg innowacji do metodologii programowania.
- ✓ Konstrukt iteracyjny ⇒ makro `loop` używane na Lisp Machines oraz w dialekcie MacLisp, aktualnie obecne w Common Lisp.

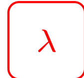
(Czesto określane jako jedna z najwiekszych wad jezyka ponieważ narusza podstawowa zasade, mówiaca o tym, że symbol powinien wystepować tylko jako pierwszy element s-wyrażenia a wszystkie pozostałe wyrażenia powinny być przedstawione jako zagnieżdżona struktura listowa. Petla `loop` to jeden z nielicznych przypadków (jedyny?) w standardzie Common Lisp, gdzie mamy doczynienia ze zorientowaniem na słowo kluczowe (*keyword-oriented*)).

Historia LISP: dialekt Interlisp

✓ Wprowadzono szereg innowacji do metodologii programowania.

✓ Konstrukt iteracyjny ⇒ makro `loop` używane na Lisp Machines oraz w dialekcie MacLisp, aktualnie obecne w Common Lisp.

(Czesto określane jako jedna z najwiekszych wad jezyka ponieważ narusza podstawowa zasade, mówiaca o tym, że symbol powinien wystepować tylko jako pierwszy element s-wyrażenia a wszystkie pozostałe wyrażenia powinny być przedstawione jako zagnieżdżona struktura listowa. Petla `loop` to jeden z nielicznych przypadków (jedyny?) w standardzie Common Lisp, gdzie mamy doczynienia ze zorientowaniem na słowo kluczowe (*keyword-oriented*)).



Historia LISP: próby standaryzacji jezyka (1)

- ✓ Pierwsze próby (1969) ⇒ Standard Lisp (podzbiór Lisp 1.5 i innych dialektów), rozszerzona implementacja ⇒ PSL (Portable Standard Lisp).
- ✓ PSL i Franz Lisp (dialekt MacLisp dla komputerów klasy Unix) ⇒ pierwsze szeroko rozpowszechnione dialekty dla różnych platform.
- ✓ Jeden z ważniejszych etapów rozwoju ⇒ druga połowa lat '70 ⇒ dialekt **Scheme** (Gerald J.Sussman i Guy L.Steele Jr.).
  - Rozszerzył LISP o koncepcje semantyki programowania z lat '60.
  - Nowości: zasieg leksykalny, kontynuacje, uproszczona składnia jezyka.

Historia LISP: próby standaryzacji jezyka (1)

✓ Pierwsze próby (1969) ⇒ Standard Lisp (podzbiór Lisp 1.5 i innych dialektów), rozszerzona implementacja ⇒ PSL (Portable Standard Lisp).

✓ PSL i Franz Lisp (dialekt MacLisp dla komputerów klasy Unix) ⇒ pierwsze szeroko rozpowszechnione dialekty dla różnych platform.

✓ Jeden z ważniejszych etapów rozwoju ⇒ druga połowa lat '70 ⇒ dialekt **Scheme** (Gerald J.Sussman i Guy L.Steele Jr.).

- Rozszerzył LISP o koncepcje semantyki programowania z lat '60.
- Nowości: zasieg leksykalny, kontynuacje, uproszczona składnia jezyka.

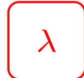
Historia LISP: próby standaryzacji jezyka (1)

✓ Pierwsze próby (1969) ⇒ Standard Lisp (podzbiór Lisp 1.5 i innych dialektów), rozszerzona implementacja ⇒ PSL (Portable Standard Lisp).

✓ PSL i Franz Lisp (dialekt MacLisp dla komputerów klasy Unix) ⇒ pierwsze szeroko rozpowszechnione dialekty dla różnych platform.

✓ Jeden z ważniejszych etapów rozwoju ⇒ druga połowa lat '70 ⇒ dialekt **Scheme** (Gerald J.Sussman i Guy L.Steele Jr.).

- Rozszerzył LISP o koncepcje semantyki programowania z lat '60.
- Nowości: zasieg leksykalny, kontynuacje, uproszczona składnia jezyka.



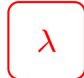
Historia LISP: próby standaryzacji jezyka (1)

✓ Pierwsze próby (1969) ⇒ Standard Lisp (podzbiór Lisp 1.5 i innych dialektów), rozszerzona implementacja ⇒ PSL (Portable Standard Lisp).

✓ PSL i Franz Lisp (dialekt MacLisp dla komputerów klasy Unix) ⇒ pierwsze szeroko rozpowszechnione dialekty dla różnych platform.

✓ Jeden z ważniejszych etapów rozwoju ⇒ druga połowa lat '70 ⇒ dialekt **Scheme** (Gerald J.Sussman i Guy L.Steele Jr.).

- Rozszerzył LISP o koncepcje semantyki programowania z lat '60.
- Nowości: zasieg leksykalny, kontynuacje, uproszczona składnia jezyka.



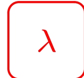
Historia LISP: próby standaryzacji jezyka (1)

✓ Pierwsze próby (1969) ⇒ Standard Lisp (podzbiór Lisp 1.5 i innych dialektów), rozszerzona implementacja ⇒ PSL (Portable Standard Lisp).

✓ PSL i Franz Lisp (dialekt MacLisp dla komputerów klasy Unix) ⇒ pierwsze szeroko rozpowszechnione dialekty dla różnych platform.

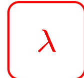
✓ Jeden z ważniejszych etapów rozwoju ⇒ druga połowa lat '70 ⇒ dialekt **Scheme** (Gerald J.Sussman i Guy L.Steele Jr.).

- Rozszerzył LISP o koncepcje semantyki programowania z lat '60.
- Nowości: zasieg leksykalny, kontynuacje, uproszczona składnia jezyka.



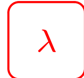
Historia LISP: próby standaryzacji jezyka (2)

- ✓ Koncepcja programowania obiektowego zaczyna wywierać bardzo silny wpływ na jezyk Lisp (późne lata '70).
- ✓ Koncepcje obiektowe z jezyka Smalltalk podstawa stworzenia LOOPS (Lisp Object Oriented Programming System) [MIT, Xerox] ⇒ Common LOOPS.
- ✓ Powstanie CLOS (Common Lisp Object System).
- ✓ Projekty Symbolics, SPICE, NIP oraz S-1 łacza siły ⇒ powstaje Common Lisp (koncepcje z MacLisp, Scheme oraz innych).
- ✓ Obowiazujacy standard ⇒ ANSI Common Lisp (1986).



Historia LISP: próby standaryzacji jezyka (2)

- ✓ Koncepcja programowania obiektowego zaczyna wywierać bardzo silny wpływ na jezyk Lisp (późne lata '70).
- ✓ Koncepcje obiektowe z jezyka Smalltalk podstawa stworzenia LOOPS (Lisp Object Oriented Programming System) [MIT, Xerox] ⇒ Common LOOPS.
- ✓ Powstanie CLOS (Common Lisp Object System).
- ✓ Projekty Symbolics, SPICE, NIP oraz S-1 łacza siły ⇒ powstaje Common Lisp (koncepcje z MacLisp, Scheme oraz innych).
- ✓ Obowiazujacy standard ⇒ ANSI Common Lisp (1986).



Historia LISP: próby standaryzacji jezyka (2)

- ✓ Koncepcja programowania obiektowego zaczyna wywierać bardzo silny wpływ na jezyk Lisp (późne lata '70).
- ✓ Koncepcje obiektowe z jezyka Smalltalk podstawa stworzenia LOOPS (Lisp Object Oriented Programming System) [MIT, Xerox] ⇒ Common LOOPS.
- ✓ Powstanie CLOS (Common Lisp Object System).
- ✓ Projekty Symbolics, SPICE, NIP oraz S-1 łacza siły ⇒ powstaje Common Lisp (koncepcje z MacLisp, Scheme oraz innych).
- ✓ Obowiazujacy standard ⇒ ANSI Common Lisp (1986).

Historia LISP: próby standaryzacji jezyka (2)

✓ Koncepcja programowania obiektowego zaczyna wywierać bardzo silny wpływ na jezyk Lisp (późne lata '70).

✓ Koncepcje obiektowe z jezyka Smalltalk podstawa stworzenia LOOPS (Lisp Object Oriented Programming System) [MIT, Xerox] ⇒ Common LOOPS.

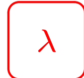
✓ Powstanie CLOS (Common Lisp Object System).

✓ Projekty Symbolics, SPICE, NIP oraz S-1 łacza siły ⇒ powstaje Common Lisp (koncepcje z MacLisp, Scheme oraz innych).

✓ Obowiazujacy standard ⇒ ANSI Common Lisp (1986).

Historia LISP: próby standaryzacji jezyka (2)

- ✓ Koncepcja programowania obiektowego zaczyna wywierać bardzo silny wpływ na jezyk Lisp (późne lata '70).
- ✓ Koncepcje obiektowe z jezyka Smalltalk podstawa stworzenia LOOPS (Lisp Object Oriented Programming System) [MIT, Xerox] ⇒ Common LOOPS.
- ✓ Powstanie CLOS (Common Lisp Object System).
- ✓ Projekty Symbolics, SPICE, NIP oraz S-1 łacza siły ⇒ powstaje Common Lisp (koncepcje z MacLisp, Scheme oraz innych).
- ✓ Obowiazujacy standard ⇒ ANSI Common Lisp (1986).



Charakterystyka jezyka LISP (według Johna McCarthy'ego)

- ✓ 1960 rok – John McCarthy publikuje artykuł *"Recursive Functions of Symbolic Expressions and Their Computation by Machine"*. Pełna specyfikacja jezyka przedstawiona na jednej stronie papieru.

- ✓ Charakterystyczna cecha ⇒ reprezentacja symboliczna. np. $x + 3y + z$ przedstawione jako `(PLUS X (TIMES 3 Y) Z)` (Tak zwana notacja *"Cambridge Polish" (polski akcent)* ponieważ przypominała prefiksowa notacje Jana Łukasiewicza).

przykładowe działanie	`[(12 + 4) - (3 / [(7 - 3) + (2 * 1 * 5)])]`
not.prefiksowa	`- + 12 4 / 3 + - 7 3 * 2 1 5`
not.Lisp/Scheme	`(- (+ 12 4) (/ 3 (+ (- 7 3) (* 2 1 5))))`

- ✓ Założenie (matematyczne): programy jako aplikowalne wyrażenia tworzone ze stałych i zmiennych przy użyciu funkcji.

Charakterystyka jezyka LISP (według Johna McCarthy'ego)

- ✓ 1960 rok – John McCarthy publikuje artykuł *"Recursive Functions of Symbolic Expressions and Their Computation by Machine"*. Pełna specyfikacja jezyka przedstawiona na jednej stronie papieru.

- ✓ Charakterystyczna cecha ⇒ reprezentacja symboliczna. np. $x + 3y + z$ przedstawione jako `(PLUS X (TIMES 3 Y) Z)` (Tak zwana notacja *"Cambridge Polish" (polski akcent)* ponieważ przypominała prefiksowa notacje Jana Łukasiewicza).

przykładowe działanie	`[(12 + 4) - (3 / [(7 - 3) + (2 * 1 * 5)])]`
not.prefiksowa	`- + 12 4 / 3 + - 7 3 * 2 1 5`
not.Lisp/Scheme	`(- (+ 12 4) (/ 3 (+ (- 7 3) (* 2 1 5))))`

- ✓ Założenie (matematyczne): programy jako aplikowalne wyrażenia tworzone ze stałych i zmiennych przy użyciu funkcji.

Charakterystyka jezyka LISP (według Johna McCarthy'ego)

- ✓ 1960 rok – John McCarthy publikuje artykuł *"Recursive Functions of Symbolic Expressions and Their Computation by Machine"*. Pełna specyfikacja jezyka przedstawiona na jednej stronie papieru.

- ✓ Charakterystyczna cecha ⇒ reprezentacja symboliczna. np. $x + 3y + z$ przedstawione jako `(PLUS X (TIMES 3 Y) Z)` (Tak zwana notacja *"Cambridge Polish" (polski akcent)* ponieważ przypominała prefiksowa notacje Jana Łukasiewicza).

przykładowe działanie	`[(12 + 4) - (3 / [(7 - 3) + (2 * 1 * 5)])]`
not.prefiksowa	`- + 12 4 / 3 + - 7 3 * 2 1 5`
not.Lisp/Scheme	`(- (+ 12 4) (/ 3 (+ (- 7 3) (* 2 1 5))))`

- ✓ Założenie (matematyczne): programy jako aplikowalne wyrażenia tworzone ze stałych i zmiennych przy użyciu funkcji.

Charakterystyka jezyka LISP (według Johna McCarthy'ego)

- ✓ 1960 rok – John McCarthy publikuje artykuł *"Recursive Functions of Symbolic Expressions and Their Computation by Machine"*. Pełna specyfikacja jezyka przedstawiona na jednej stronie papieru.

- ✓ Charakterystyczna cecha ⇒ reprezentacja symboliczna. np. $x + 3y + z$ przedstawione jako `(PLUS X (TIMES 3 Y) Z)` (Tak zwana notacja *"Cambridge Polish" (polski akcent)* ponieważ przypominała prefiksowa notacje Jana Łukasiewicza).

przykładowe działanie	`[(12 + 4) - (3 / [(7 - 3) + (2 * 1 * 5)])]`
not.prefiksowa	`- + 12 4 / 3 + - 7 3 * 2 1 5`
not.Lisp/Scheme	`(- (+ 12 4) (/ 3 (+ (- 7 3) (* 2 1 5))))`

- ✓ Założenie (matematyczne): programy jako aplikowalne wyrażenia tworzone ze stałych i zmiennych przy użyciu funkcji.

Charakterystyka jezyka LISP (powstanie interpretera)

- ✓ Każda operacje przedstawiona przy pomocy Maszyny Turinga można w prosty sposób przedstawić za pomoca działań w Lispie.

- ✓ Cheć wykazania wyższości Lispa nad sposobem reprezentacji funkcji oraz ogólnych definicji rekurencyjnych (używanych w teorii funkcji) Maszyna Turinga.

- ✓ Powstanie uniwersalnej funkcji **eval[e,a]** obliczajacej wyrażenia **e** (gdzie **a** jest lista przypisań wartości do zmiennych $\rightarrow$ potrzebne do działań rekurencyjnych).

- ✓ Funkcja **eval** $\Rightarrow$ notacja reprezentujaca programy jako dane (konstrukt `lambda` analogiczny do operatora λ Alonzo Churcha [rachunek λ-Churcha]).

- ✓ Wielka niespodzianka $\Rightarrow$ **eval** może służyć jako interpreter dla jezyka Lisp.

Charakterystyka jezyka LISP (powstanie interpretera)

- ✓ Każda operacje przedstawiona przy pomocy Maszyny Turinga można w prosty sposób przedstawić za pomoca działań w Lispie.
- ✓ Cheć wykazania wyższości Lispa nad sposobem reprezentacji funkcji oraz ogólnych definicji rekurencyjnych (używanych w teorii funkcji) Maszyna Turinga.
- ✓ Powstanie uniwersalnej funkcji **eval[e,a]** obliczajacej wyrażenia **e** (gdzie **a** jest lista przypisań wartości do zmiennych $\rightarrow$ potrzebne do działań rekurencyjnych).
- ✓ Funkcja **eval** $\Rightarrow$ notacja reprezentujaca programy jako dane (konstrukt `lambda` analogiczny do operatora λ Alonzo Churcha [rachunek λ-Churcha]).
- ✓ Wielka niespodzianka $\Rightarrow$ **eval** może służyć jako interpreter dla jezyka Lisp.

Charakterystyka jezyka LISP (powstanie interpretera)

- ✓ Każda operacje przedstawiona przy pomocy Maszyny Turinga można w prosty sposób przedstawić za pomoca działań w Lispie.
- ✓ Cheć wykazania wyższości Lispa nad sposobem reprezentacji funkcji oraz ogólnych definicji rekurencyjnych (używanych w teorii funkcji) Maszyna Turinga.
- ✓ Powstanie uniwersalnej funkcji **eval[e,a]** obliczajacej wyrażenia **e** (gdzie **a** jest lista przypisań wartości do zmiennych → potrzebne do działań rekurencyjnych).
- ✓ Funkcja **eval** ⇒ notacja reprezentujaca programy jako dane (konstrukt `lambda` analogiczny do operatora λ Alonzo Churcha [rachunek λ-Churcha]).
- ✓ Wielka niespodzianka ⇒ **eval** może służyć jako interpreter dla jezyka Lisp.

Charakterystyka jezyka LISP (powstanie interpretera)

- ✓ Każda operacje przedstawiona przy pomocy Maszyny Turinga można w prosty sposób przedstawić za pomoca działań w Lispie.

- ✓ Cheć wykazania wyższości Lispa nad sposobem reprezentacji funkcji oraz ogólnych definicji rekurencyjnych (używanych w teorii funkcji) Maszyna Turinga.

- ✓ Powstanie uniwersalnej funkcji **eval[e,a]** obliczajacej wyrażenia **e** (gdzie **a** jest lista przypisań wartości do zmiennych → potrzebne do działań rekurencyjnych).

- ✓ Funkcja **eval** ⇒ notacja reprezentujaca programy jako dane (konstrukt `lambda` analogiczny do operatora λ Alonzo Churcha [rachunek λ-Churcha]).

- ✓ Wielka niespodzianka ⇒ **eval** może służyć jako interpreter dla jezyka Lisp.

Charakterystyka jezyka LISP (powstanie interpretera)

- ✓ Każda operacje przedstawiona przy pomocy Maszyny Turinga można w prosty sposób przedstawić za pomoca działań w Lispie.
- ✓ Cheć wykazania wyższości Lispa nad sposobem reprezentacji funkcji oraz ogólnych definicji rekurencyjnych (używanych w teorii funkcji) Maszyna Turinga.
- ✓ Powstanie uniwersalnej funkcji **eval[e,a]** obliczajacej wyrażenia **e** (gdzie **a** jest lista przypisań wartości do zmiennych $\rightarrow$ potrzebne do działań rekurencyjnych).
- ✓ Funkcja **eval** $\Rightarrow$ notacja reprezentujaca programy jako dane (konstrukt `lambda` analogiczny do operatora λ Alonzo Churcha [rachunek λ-Churcha]).
- ✓ Wielka niespodzianka $\Rightarrow$ **eval** może służyć jako interpreter dla jezyka Lisp.

Mocne strony jezyka

Lisp to drugi (po FORTRANie) z najstarszych jezyków programowania szeroko rozpowszechniony w użyciu, swoja długowieczność zawdziecza dwóm faktom:

❶ Dobrze zdefiniowany rdzeń jezyka, rekursywne użycie wyrażeń warunkowych, reprezentacja informacji symbolicznych (w tym programów) za pomoca list.

❷ Niedoścignione cechy sprawiajace że jest użytecznym jezykiem do obliczeń symbolicznych na wyższym poziomie oraz do AI, wbudowany interpreter.

Lisp przetrwał ponieważ programy w nim pisane sa listami, mimo iż wszyscy – właczajac Johna McCarthy'ego – uważali ten fakt za poważna wade (!)

Mocne strony jezyka

Lisp to drugi (po FORTRANie) z najstarszych jezyków programowania szeroko rozpowszechniony w użyciu, swoja długowieczność zawdzięcza dwóm faktom:

❶ Dobrze zdefiniowany rdzeń jezyka, rekursywne użycie wyrażeń warunkowych, reprezentacja informacji symbolicznych (w tym programów) za pomoca list.

❷ Niedoścignione cechy sprawiajace że jest użytecznym jezykiem do obliczeń symbolicznych na wyższym poziomie oraz do AI, wbudowany interpreter.

Lisp przetrwał ponieważ programy w nim pisane sa listami, mimo iż wszyscy – właczajac Johna McCarthy'ego – uważali ten fakt za poważna wade (!)

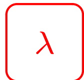
Co sprawiło, że Lisp był (jest) inny?

Odrebność jezyka Lisp wynikała bezpośrednio z 9 nowych koncepcji które pojawiły sie wraz z tym jezykiem.

❶ **Wyrażenia warunkowe.** (konstrukt `if-then-else`) wynaleziony przez McCarthy'ego w toku prac nad jezykiem. (FORTRAN dysponował jedynie warunkowa instrukcja `goto` blisko zwiazana z instrukcjami sprzetowymi). McCarthy ⇒ Algol ⇒ rozpowszechnienie instrukcji warunkowych.

❷ **Funkcja jako typ danych.** Funkcje jako obiekty pierwszej klasy (typy danych tak jak `integer`, `string` itp.). Maja swoja reprezentacje, moga być przechowywane w zmiennych, przekazywane jako argumenty itp.

❸ **Rekursja.** Wcześniej ⇒ koncepcja matematyczna, LISP jako pierwszy jezyk w pełni wspierał definicje rekurencyjne.



Co sprawiło, że Lisp był (jest) inny?

Odrebność jezyka Lisp wynikała bezpośrednio z 9 nowych koncepcji które pojawiły sie wraz z tym jezykiem.

❶ **Wyrażenia warunkowe.** (konstrukt `if-then-else`) wynaleziony przez McCarthy'ego w toku prac nad jezykiem. (FORTRAN dysponował jedynie warunkowa instrukcja `goto` blisko zwiazana z instrukcjami sprzetowymi). McCarthy ⇒ Algol ⇒ rozpowszechnienie instrukcji warunkowych.

❷ **Funkcja jako typ danych.** Funkcje jako obiekty pierwszej klasy (typy danych tak jak `integer`, `string` itp.). Maja swoja reprezentacje, moga być przechowywane w zmiennych, przekazywane jako argumenty itp.

❸ **Rekursja.** Wcześniej ⇒ koncepcja matematyczna, LISP jako pierwszy jezyk w pełni wspierał definicje rekurencyjne.

Co sprawiło, że Lisp był (jest) inny?

Odrebność jezyka Lisp wynikała bezpośrednio z 9 nowych koncepcji które pojawiły sie wraz z tym jezykiem.

❶ **Wyrażenia warunkowe.** (konstrukt `if-then-else`) wynaleziony przez McCarthy'ego w toku prac nad jezykiem. (FORTRAN dysponował jedynie warunkowa instrukcja `goto` blisko zwiazana z instrukcjami sprzetowymi). McCarthy ⇒ Algol ⇒ rozpowszechnienie instrukcji warunkowych.

❷ **Funkcja jako typ danych.** Funkcje jako obiekty pierwszej klasy (typy danych tak jak `integer`, `string` itp.). Maja swoja reprezentacje, moga być przechowywane w zmiennych, przekazywane jako argumenty itp.

❸ **Rekursja.** Wcześniej ⇒ koncepcja matematyczna, LISP jako pierwszy jezyk w pełni wspierał definicje rekurencyjne.

❹ **Nowa koncepcja zmiennych.** Wszystkie zmienne sa wskaźnikami. Przypisywanie do zmiennych (symboli) ⇒ kopiowanie wskaźników (a nie tego na co wskazuja).

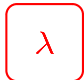
❺ **Garbage Collector.** Automatyczne zarzadzanie pamiecia.

❻ **Programy złożone z wyrażeń.** Programy ⇒ drzewa wyrażeń zwracajacych wartości. (inaczej niż np. w jezyku FORTRAN ⇒ rozróżnienie miedzy wyrażeniami a deklaracjami). Dowolność w konstruowaniu wyrażeń.

```
(if foo (= x 1) (= x 2))
          lub
(= x (if foo 1 2))
```

❼ **Symbol jako typ danych.** Różnica w porównaniu do np. stringów ⇒ porównywania przy użyciu wskaźników.

❽ **Notacja złożona z drzew symboli.**



❹ **Nowa koncepcja zmiennych.** Wszystkie zmienne sa wskaźnikami. Przypisywanie do zmiennych (symboli) ⇒ kopiowanie wskaźników (a nie tego na co wskazuja).

❺ **Garbage Collector.** Automatyczne zarzadzanie pamiecia.

❻ **Programy złożone z wyrażeń.** Programy ⇒ drzewa wyrażeń zwracajacych wartości. (inaczej niż np. w jezyku FORTRAN ⇒ rozróżnienie miedzy wyrażeniami a deklaracjami). Dowolność w konstruowaniu wyrażeń.

```
(if foo (= x 1) (= x 2))
         lub
(= x (if foo 1 2))
```

❼ **Symbol jako typ danych.** Różnica w porównaniu do np. stringów ⇒ porównywania przy użyciu wskaźników.

❽ **Notacja złożona z drzew symboli.**

❹ **Nowa koncepcja zmiennych.** Wszystkie zmienne sa wskaźnikami. Przypisywanie do zmiennych (symboli) ⇒ kopiowanie wskaźników (a nie tego na co wskazuja).

❺ **Garbage Collector.** Automatyczne zarzadzanie pamiecia.

❻ **Programy złożone z wyrażeń.** Programy ⇒ drzewa wyrażeń zwracajacych wartości. (inaczej niż np. w jezyku FORTRAN ⇒ rozróżnienie miedzy wyrażeniami a deklaracjami). Dowolność w konstruowaniu wyrażeń.

```
(if foo (= x 1) (= x 2))
         lub
(= x (if foo 1 2))
```

❼ **Symbol jako typ danych.** Różnica w porównaniu do np. stringów ⇒ porównywania przy użyciu wskaźników.

❽ **Notacja złożona z drzew symboli.**

❹ **Nowa koncepcja zmiennych.** Wszystkie zmienne sa wskaźnikami. Przypisywanie do zmiennych (symboli) ⇒ kopiowanie wskaźników (a nie tego na co wskazuja).

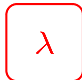
❺ **Garbage Collector.** Automatyczne zarzadzanie pamiecia.

❻ **Programy złożone z wyrażeń.** Programy ⇒ drzewa wyrażeń zwracajacych wartości. (inaczej niż np. w jezyku FORTRAN ⇒ rozróżnienie miedzy wyrażeniami a deklaracjami). Dowolność w konstruowaniu wyrażeń.

```
(if foo (= x 1) (= x 2))
          lub
(= x (if foo 1 2))
```

❼ **Symbol jako typ danych.** Różnica w porównaniu do np. stringów ⇒ porównywania przy użyciu wskaźników.

❽ **Notacja złożona z drzew symboli.**



❹ **Nowa koncepcja zmiennych.** Wszystkie zmienne sa wskaźnikami. Przypisywanie do zmiennych (symboli) ⇒ kopiowanie wskaźników (a nie tego na co wskazuja).

❺ **Garbage Collector.** Automatyczne zarzadzanie pamiecia.

❻ **Programy złożone z wyrażeń.** Programy ⇒ drzewa wyrażeń zwracajacych wartości. (inaczej niż np. w jezyku FORTRAN ⇒ rozróżnienie miedzy wyrażeniami a deklaracjami). Dowolność w konstruowaniu wyrażeń.

```
(if foo (= x 1) (= x 2))
        lub
(= x (if foo 1 2))
```

❼ **Symbol jako typ danych.** Różnica w porównaniu do np. stringów ⇒ porównywania przy użyciu wskaźników.

❽ **Notacja złożona z drzew symboli.**

❾ Dowolność w dostepie do możliwości oferowanych przez jezyk.

Nie istnieje żadna rzeczywista różnica miedzy okresem w którym program jest **wczytywany, kompilowany i uruchamiany**. Z poziomu jezyka można **kompilować lub uruchamiać** kod podczas **wczytywania**, **wczytywać lub uruchamiać** kod podczas **kompilacji** oraz **wczytywać lub kompilować** kod **w czasie działania** programu.

- Uruchamianie kodu w czasie wczytywania umożliwia reprogramowanie składni jezyka.
- Uruchamianie kodu w czasie kompilacji jest podstawa definiowania makr.
- Kompilowanie w czasie uruchamiania jest podstawa wykorzystania jezyka LISP jako jezyka rozszerzeń (np. Emacs Lisp).
- Wczytywanie podczas uruchamiania umożliwia programom komunikowanie sie za pomoca s-wyrażeń (koncepcja ostatnio na nowo odkryta pod nazwa XML ;-)).

❾ Dowolność w dostepie do możliwości oferowanych przez jezyk.

Nie istnieje żadna rzeczywista różnica miedzy okresem w którym program jest **wczytywany, kompilowany i uruchamiany**. Z poziomu jezyka można **kompilować lub uruchamiać** kod podczas **wczytywania**, **wczytywać lub uruchamiać** kod podczas **kompilacji** oraz **wczytywać lub kompilować** kod **w czasie działania** programu.

- Uruchamianie kodu w czasie wczytywania umożliwia reprogramowanie składni jezyka.
- Uruchamianie kodu w czasie kompilacji jest podstawa definiowania makr.
- Kompilowanie w czasie uruchamiania jest podstawa wykorzystania jezyka LISP jako jezyka rozszerzeń (np. Emacs Lisp).
- Wczytywanie podczas uruchamiania umożliwia programom komunikowanie sie za pomoca s-wyrażeń (koncepcja ostatnio na nowo odkryta pod nazwa XML ;-)).

❾ Dowolność w dostepie do możliwości oferowanych przez jezyk.

Nie istnieje żadna rzeczywista różnica miedzy okresem w którym program jest **wczytywany, kompilowany i uruchamiany**. Z poziomu jezyka można **kompilować lub uruchamiać** kod podczas **wczytywania**, **wczytywać lub uruchamiać** kod podczas **kompilacji** oraz **wczytywać lub kompilować** kod **w czasie działania** programu.

- Uruchamianie kodu w czasie wczytywania umożliwia reprogramowanie składni jezyka.
- Uruchamianie kodu w czasie kompilacji jest podstawa definiowania makr.
- Kompilowanie w czasie uruchamiania jest podstawa wykorzystania jezyka LISP jako jezyka rozszerzeń (np. Emacs Lisp).
- Wczytywanie podczas uruchamiania umożliwia programom komunikowanie sie za pomoca s-wyrażeń (koncepcja ostatnio na nowo odkryta pod nazwa XML ;-)).

❾ Dowolność w dostepie do możliwości oferowanych przez jezyk.

Nie istnieje żadna rzeczywista różnica miedzy okresem w którym program jest **wczytywany, kompilowany i uruchamiany**. Z poziomu jezyka można **kompilować lub uruchamiać** kod podczas **wczytywania**, **wczytywać lub uruchamiać** kod podczas **kompilacji** oraz **wczytywać lub kompilować** kod **w czasie działania** programu.

- Uruchamianie kodu w czasie wczytywania umożliwia reprogramowanie składni jezyka.
- Uruchamianie kodu w czasie kompilacji jest podstawa definiowania makr.
- Kompilowanie w czasie uruchamiania jest podstawa wykorzystania jezyka LISP jako jezyka rozszerzeń (np. Emacs Lisp).
- Wczytywanie podczas uruchamiania umożliwia programom komunikowanie sie za pomoca s-wyrażeń (koncepcja ostatnio na nowo odkryta pod nazwa XML ;-)).

❾ Dowolność w dostepie do możliwości oferowanych przez jezyk.

Nie istnieje żadna rzeczywista różnica miedzy okresem w którym program jest **wczytywany, kompilowany i uruchamiany**. Z poziomu jezyka można **kompilować lub uruchamiać** kod podczas **wczytywania**, **wczytywać lub uruchamiać** kod podczas **kompilacji** oraz **wczytywać lub kompilować** kod **w czasie działania** programu.

- Uruchamianie kodu w czasie wczytywania umożliwia reprogramowanie składni jezyka.
- Uruchamianie kodu w czasie kompilacji jest podstawa definiowania makr.
- Kompilowanie w czasie uruchamiania jest podstawa wykorzystania jezyka LISP jako jezyka rozszerzeń (np. Emacs Lisp).
- Wczytywanie podczas uruchamiania umożliwia programom komunikowanie sie za pomoca s-wyrażeń (koncepcja ostatnio na nowo odkryta pod nazwa XML ;-)).

Kierunki rozwoju jezyków programowania

Punkty ❶–❺ wystepuja we współczesnych jezykach. Koncepcja ❻ powoli zaczyna zyskiwać na znaczeniu. Jezyk Python ma strukture podobna do ❼. Punkty ❽–❾ sa wciaż niedoścignione i specyficzne dla jezyka Lisp, ponieważ:

✓ Wymagaja do działania użycia nawiasów lub podobnej koncepcji.

✓ Pojawienie sie tego typu rozszerzeń nie oznacza, że wynaleziono nowy jezyk, można jedynie mówić o nowym dialekcie jezyka Lisp ;-)

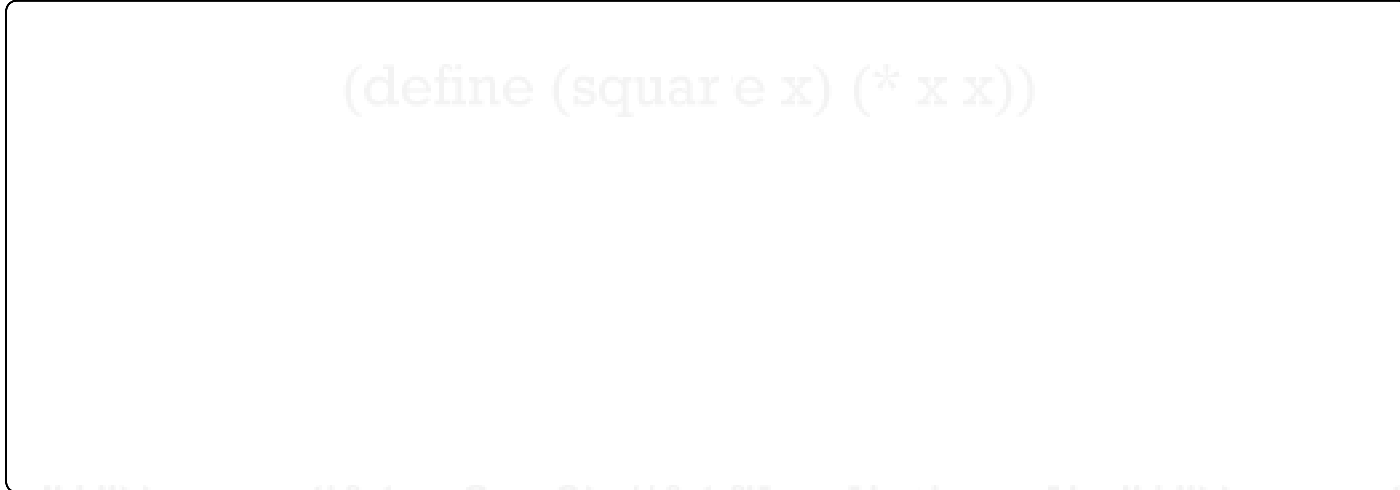
Kierunki rozwoju jezyków programowania

Punkty ❶–❺ wystepuja we współczesnych jezykach. Koncepcja ❻ powoli zaczyna zyskiwać na znaczeniu. Jezyk Python ma strukture podobna do ❼. Punkty ❽–❾ sa wciaż niedoścignione i specyficzne dla jezyka Lisp, ponieważ:

- ✓ Wymagaja do działania użycia nawiasów lub podobnej koncepcji.
- ✓ Pojawienie sie tego typu rozszerzeń nie oznacza, że wynaleziono nowy jezyk, można jedynie mówić o nowym dialekcie jezyka Lisp ;-)

Kolejność przetwarzania wyrażeń

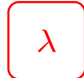
Zgodnie z notacja polska (prefiksowa) Jana Łukasiewicza, interpreter jezyka Lisp rozpocznie przetwarzanie wyrażenia arytmetycznego poczynajac od operatora a nastepnie jego argumentów (od lewej do prawej). Jeśli wynik wyrażenia nie może być bezpośrednio określony (np. wyrażenie zawiera podwyrażenia) nastapi najpierw obliczenie wartości argumentów bedacych podwyrażeniami.



Kolejność przetwarzania wyrażeń

Zgodnie z notacja polska (prefiksowa) Jana Łukasiewicza, interpreter jezyka Lisp rozpocznie przetwarzanie wyrażenia arytmetycznego poczynajac od operatora a nastepnie jego argumentów (od lewej do prawej). Jeśli wynik wyrażenia nie może być bezpośrednio określony (np. wyrażenie zawiera podwyrażenia) nastapi najpierw obliczenie wartości argumentów bedacych podwyrażeniami.

```
❶ (+ (* (- 10 7) (+ 4 1)) (- 15 (/ 12 3)) 17)
```



Kolejność przetwarzania wyrażeń

Zgodnie z notacja polska (prefiksowa) Jana Łukasiewicza, interpreter jezyka Lisp rozpocznie przetwarzanie wyrażenia arytmetycznego poczynajac od operatora a nastepnie jego argumentów (od lewej do prawej). Jeśli wynik wyrażenia nie może być bezpośrednio określony (np. wyrażenie zawiera podwyrażenia) nastapi najpierw obliczenie wartości argumentów bedacych podwyrażeniami.

```
❶ (+ (* (- 10 7) (+ 4 1)) (- 15 (/ 12 3)) 17)
❷ (+ (* 3 (+ 4 1)) (- 15 (/ 12 3)) 17)
```

Kolejność przetwarzania wyrażeń

Zgodnie z notacja polska (prefiksowa) Jana Łukasiewicza, interpreter jezyka Lisp rozpocznie przetwarzanie wyrażenia arytmetycznego poczynajac od operatora a nastepnie jego argumentów (od lewej do prawej). Jeśli wynik wyrażenia nie może być bezpośrednio określony (np. wyrażenie zawiera podwyrażenia) nastapi najpierw obliczenie wartości argumentów bedacych podwyrażeniami.

```
❶ (+ (* (- 10 7) (+ 4 1)) (- 15 (/ 12 3)) 17)
❷ (+ (* 3 (+ 4 1)) (- 15 (/ 12 3)) 17)
❸ (+ (* 3 5 ) (- 15 (/ 12 3)) 17)
```

Kolejność przetwarzania wyrażeń

Zgodnie z notacja polska (prefiksowa) Jana Łukasiewicza, interpreter jezyka Lisp rozpocznie przetwarzanie wyrażenia arytmetycznego poczynajac od operatora a nastepnie jego argumentów (od lewej do prawej). Jeśli wynik wyrażenia nie może być bezpośrednio określony (np. wyrażenie zawiera podwyrażenia) nastapi najpierw obliczenie wartości argumentów bedacych podwyrażeniami.

```
❶ (+ (* (- 10 7) (+ 4 1)) (- 15 (/ 12 3)) 17)
❷ (+ (* 3 (+ 4 1)) (- 15 (/ 12 3)) 17)
❸ (+ (* 3 5 ) (- 15 (/ 12 3)) 17)
❹ (+ 15 (- 15 (/ 12 3)) 17)
```

Kolejność przetwarzania wyrażeń

Zgodnie z notacja polska (prefiksowa) Jana Łukasiewicza, interpreter jezyka Lisp rozpocznie przetwarzanie wyrażenia arytmetycznego poczynajac od operatora a nastepnie jego argumentów (od lewej do prawej). Jeśli wynik wyrażenia nie może być bezpośrednio określony (np. wyrażenie zawiera podwyrażenia) nastapi najpierw obliczenie wartości argumentów bedacych podwyrażeniami.

```
❶ (+ (* (- 10 7) (+ 4 1)) (- 15 (/ 12 3)) 17)
❷ (+ (* 3 (+ 4 1)) (- 15 (/ 12 3)) 17)
❸ (+ (* 3 5 ) (- 15 (/ 12 3)) 17)
❹ (+ 15 (- 15 (/ 12 3)) 17)
❺ (+ 15 (- 15 4 ) 17)
```

Kolejność przetwarzania wyrażeń

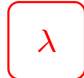
Zgodnie z notacja polska (prefiksowa) Jana Łukasiewicza, interpreter jezyka Lisp rozpocznie przetwarzanie wyrażenia arytmetycznego poczynajac od operatora a nastepnie jego argumentów (od lewej do prawej). Jeśli wynik wyrażenia nie może być bezpośrednio określony (np. wyrażenie zawiera podwyrażenia) nastapi najpierw obliczenie wartości argumentów bedacych podwyrażeniami.

```
❶ (+ (* (- 10 7) (+ 4 1)) (- 15 (/ 12 3)) 17)
❷ (+ (* 3 (+ 4 1)) (- 15 (/ 12 3)) 17)
❸ (+ (* 3 5 ) (- 15 (/ 12 3)) 17)
❹ (+ 15 (- 15 (/ 12 3)) 17)
❺ (+ 15 (- 15 4 ) 17)
❻ (+ 15 11 17)
```

Kolejność przetwarzania wyrażeń

Zgodnie z notacja polska (prefiksowa) Jana Łukasiewicza, interpreter jezyka Lisp rozpocznie przetwarzanie wyrażenia arytmetycznego poczynajac od operatora a nastepnie jego argumentów (od lewej do prawej). Jeśli wynik wyrażenia nie może być bezpośrednio określony (np. wyrażenie zawiera podwyrażenia) nastapi najpierw obliczenie wartości argumentów bedacych podwyrażeniami.

```
❶ (+ (* (- 10 7) (+ 4 1)) (- 15 (/ 12 3)) 17)
❷ (+ (* 3 (+ 4 1)) (- 15 (/ 12 3)) 17)
❸ (+ (* 3 5 ) (- 15 (/ 12 3)) 17)
❹ (+ 15 (- 15 (/ 12 3)) 17)
❺ (+ 15 (- 15 4 ) 17)
❻ (+ 15 11 17)
❼ 43
```



Struktury i typy danych w jezyku Lisp/Scheme

```
atom        →  liczba | symbol | ciag_znaków
symbol      →  dowolna_sekwencja_niezarezerwowanych_znaków
pozycja     →  atom | lista
sekwencja   →  lista_pusta | pozycja | sekwencja
lista       →  (sekwencja)
```

Struktury i typy danych w jezyku Lisp/Scheme

```
atom        →  liczba | symbol | ciag_znaków
symbol      →  dowolna_sekwencja_niezarezerwowanych_znaków
pozycja     →  atom | lista
sekwencja   →  lista_pusta | pozycja | sekwencja
lista       →  (sekwencja)
```

JEZYK	PRAWDA	FAŁSZ
LISP	t, T	nil, NIL, ()
SCHEME	#t	#f, ()

Budowa i reprezentacja struktury listowej (lista a para)

Notacja "klasyczna".

```
(a b c d) ; poprawna lista zlozona z 4 elementów
```

Notacja "kropkowa" (tzw. dotted-pair).

```
(a . (b . (c . (d . ())))) ; alternatywny sposób zapisu
```

Para (struktura danych) postaci `(x . y)` to nieprawidłowa/niepełna lista (nie kończy sie elementem NIL). Każda liste można przedstawić jako zagnieżdżone pary w notacji kropkowej. Element znajdujacy sie przed kropka (w notacji kropkowej) nazywamy CAR, element znajdujacy sie po kropce nazywamy CDR.

Budowa i reprezentacja struktury listowej (lista a para)

Notacja "klasyczna".

```
(a b c d) ; poprawna lista zlozona z 4 elementów
```

Notacja "kropkowa" (tzw. dotted-pair).

```
(a . (b . (c . (d . ())))) ; alternatywny sposób zapisu
```

Para (struktura danych) postaci `(x . y)` to nieprawidłowa/niepełna lista (nie kończy sie elementem NIL). Każda liste można przedstawić jako zagnieżdżone pary w notacji kropkowej. Element znajdujacy sie przed kropka (w notacji kropkowej) nazywamy CAR, element znajdujacy sie po kropce nazywamy CDR.

Budowa i reprezentacja struktury listowej (lista a para)

Notacja "klasyczna".

```
(a b c d) ; poprawna lista zlozona z 4 elementów
```

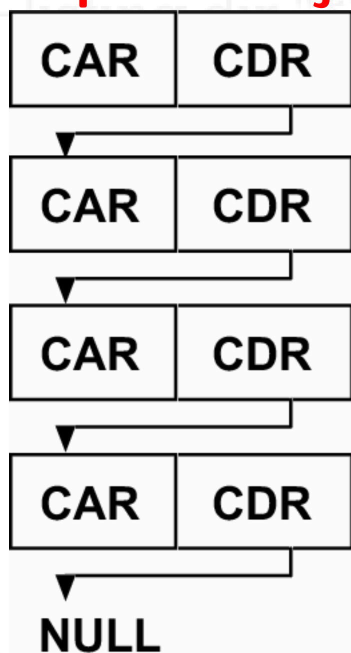
Notacja "kropkowa" (tzw. dotted-pair).

```
(a . (b . (c . (d . ())))) ; alternatywny sposób zapisu
```

Para (struktura danych) postaci `(x . y)` to nieprawidłowa/niepełna lista (nie kończy sie elementem NIL). Każda liste można przedstawić jako zagnieżdżone pary w notacji kropkowej. Element znajdujacy sie przed kropka (w notacji kropkowej) nazywamy CAR, element znajdujacy sie po kropce nazywamy CDR.

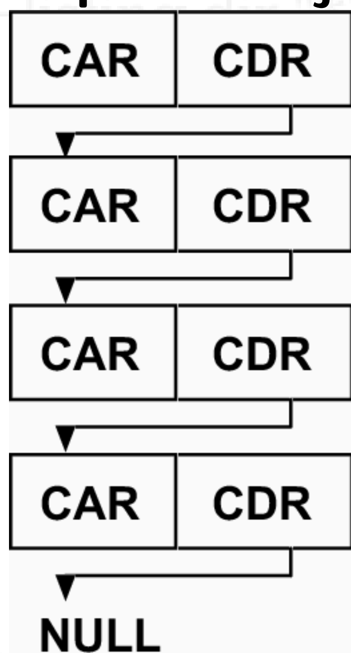
file−listing dir "*")) (if (eq? n 0) #f (:file−listing dir "*")) (if (eq? n 0) #f
(cond ((null? ls) '()) ((pred (car ls)) (cons (car ls) (filter pred (cdr ls))) (else (filter pred (cdr ls)))))
e x) (* x x)) (define (squar e x) (* x x)) (define (squa

Reprezentacja listy w pamieci komputera.



Liste `(a b c d)` (zapisana w postaci kropkowej jako `(a . (b . (c . (d . ()))))`) możemy przedstawić w pamieci komputera jako powiazane struktury złożone z elementów CAR i CDR gdzie ostatni CDR wskazuje na liste pusta `()`.

Reprezentacja listy w pamieci komputera.



Liste `(a b c d)` (zapisana w postaci kropkowej jako `(a . (b . (c . (d . ()))))` możemy przedstawić w pamieci komputera jako powiazane struktury złożone z elementów CAR i CDR gdzie ostatni CDR wskazuje na liste pusta `()`.

Reprezentacja pary w pamieci komputera.

Para (niepoprawna lista) w pamieci wystepuje jako pojedynczy CONS.

```
(a . b) ; zlozenie (cons 'a 'b)
```

Siedem podstawowych operatorów w jezyku LISP

Czym sa wyrażenia w jezyku Lisp?
Wyrażeniem w jezyku Lisp nazywamy *atom* bedacy sekwencja znaków (np. `foo`) lub *liste* złożona z 0 lub wiekszej ilości *wyrażeń* oddzielonych znakiem spacji i umieszczonych w nawiasach.

```
foo
()
(foo)
(foo bar)
(a b (c) d)
```

Wszystkie wyrażenia w Lispie zwracaja wartość. Jeśli wyrażenie jest lista, pierwszy element nazywamy *operatorem* a pozostałe *argumentami*. W pierwotnej wersji jezyka Lisp zdefiniowano siedem operatorów: `quote`, `atom`, `eq`, `car`, `cdr`, `cons` oraz `cond`. Wszystkie operatory za wyjatkiem `quote` oraz `cond` przetwarzaja swoje argumenty natychmiast po wywołaniu. Tego typu operatory nazywamy funkcjami.

Powstrzymywanie interpretera przed przetwarzaniem wyrażeń

❶ `(quote` x`)` zwraca x. W skrócie `(quote` x`)` zapisujemy jako `'`x.

Operator `quote` jest specyficzny dla jezyka Lisp. Ponieważ kod i dane składaja sie z takich samych struktur, `quote` umożliwia rozróżnianie pomiedzy nimi.

```
> (quote a)
a
> 'a
a
> (quote (a b c d))
(a b c d)
```

Sprawdzanie czy argument jest atomem

❷ (atom x) zwraca atom t jeśli wartość x jest atomem lub liste pusta ()
w przeciwnym wypadku. Wartość t reprezentuje prawde a lista pusta () fałsz.

```
> (atom 'a)
t
> (atom '(a b c d))
()
> (atom '())
t
```

Teraz dysponujac operatorem którego argument zostanie przetworzony, możemy pokazać do czego służy quote. Używajac quote na liście chronimy ja przed przetworzeniem, w przeciwnym przypadku lista przekazana jako *argument* do *operatora* takiego jak atom traktowana jest jako kod.

```
> (atom (atom 'a))
t
```

```
> (atom '(atom 'a))
()
```

podczas gdy lista poprzedzona znakiem ' nie podlega przetworzeniu.

Sprawdzenie czy argumenty sa tym samym atomem

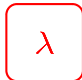
❸ (eq x y) zwraca t gdy wartości x oraz y sa tym samym atomem lub listami pustymi, w przeciwnym razie zwraca ().

```
> (eq 'a 'a)
t
> (eq 'a 'b)
()
> (eq '() '())
t
```

Pobranie pierwszego elementu listy

❹ (car x) oczekuje, że x bedzie lista i zwraca jej pierwszy element.

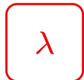
```
> (car '(a b c d))
a
```



Pobranie pozostałych elementów listy za wyjatkiem pierwszego

❺ `(cdr` x`)` oczekuje, że x bedzie lista i zwraca wszystko poza pierwszym elementem.

```
> (cdr '(a b c d))
(b c d)
```



Doklejenie elementu na poczatek listy

❻ (cons x y) oczekuje, że x bedzie lista i zwraca liste złożona z elementów listy y nastepujacych po x.

```
> (cons 'a '(b c d))
(a b c d)
> (cons 'a (cons 'b (cons 'c '())))
(a b c d)
> (car (cons 'a '(b c d)))
a
> (cdr (cons 'a '(b c d)))
(b c d)
```

Instrukcja warunkowa `cond`

❼ `(cond` $(p_1 \quad e_1)$ `...` $(p_n \quad e_n)$`)` to instrukcja warunkowa. Wyrażenia p podlegaja przetworzeniu w kolejności aż do momentu, gdy któreś z nich zwróci wartość `t`. Instrukcja `cond` zwraca wartość przetworzenia wyrażenia e odpowiadajacego warunkowi p.

```
> (cond ((eq 'a 'b) 'first)
        ((atom 'a)  'second))
second
```

Notacja funkcyjna

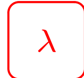
Definiowanie procedur nienazwanych

Funkcje zapisujemy jako `((lambda (`$p_1 \; \dots \; p_n$`) `e`) `$a_1 \; \dots \; a_n$`)`, gdzie $p_1 \; \dots \; p_n$ to atomy (zwane *parametrami*) a e to wyrażenie. Wyrażenie postaci

`((lambda (`$p_1 \; \dots \; p_n$`) `e`) `$a_1 \; \dots \; a_n$`)`

nazywamy *wywołaniem funkcji*. Wyrażenia a_i zostaja przetworzone. Podczas przetwarzania wyrażenia e wartość dowolnego wystapienia jednego z p_i jest wartościa odpowiadajacego mu a_i w ostatnim wywołaniu funkcji.

```
> ((lambda (x) (cons x '(b c d))) 'a)
(a b c d)
> ((lambda (x y) (cons x (cdr y)))
   'z
   '(a b c d))
(z b c d)
```



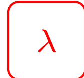
Parametry moga zostać użyte jako operatory w wyrażeniu lub jako argumenty*.

```
> ((lambda (f) (<funcall> f '(b c d)))
   '(lambda (x) (cons 'a x)))
(a b c d)
```

Inne przydatne funkcje wbudowane

Poza siedmioma wbudowanymi operatorami wystepuja także funkcje skrótu dla czesto wystepujacych przypadków. Funkcja cxr gdzie x jest sekwencja liter a i/lub d odpowiada złożeniom wywołań funkcji car oraz cdr. Implementacje jezyka Lisp na ogół gwarantuja wystepowanie tego rodzaju skrótów do 4 poziomów zagnieżdżeń.

```
> (cadr '((a b) (c d) e))
(c d)
> (caddr '((a b) (c d) e))
e
> (cdar '((a b) (c d) e))
(b)
```



Składanie list z elementów

Alternatywna metoda konstruowania list w stosunku do `cons` jest funkcja `list`.

```
> (cons 'a (cons 'b (cons 'c (cons 'd '()))))
(a b c d)
> (list 'a 'b 'c 'd)
(a b c d)
```

Definiowanie własnych funkcji

Sprawdzanie czy argument jest lista pusta

① (null. x) sprawdza czy argument jest lista pusta.

```
(defun null. (x)
  (eq x '()))
> (null. 'a)
()
> (null. '())
t
```

file–listing dir "*")) (if (eq? n 0) #f (:file–listing dir "*")) (if (eq? n 0) #f
(cond ((null? ls) '()) ((pred (car ls)) (cons (car ls) (filter pred (cdr ls)))) (else (filter pred (cdr ls)))))
(define (square x) (* x x))

Operator logiczny AND

② (and. x y) zwraca t gdy argumenty sa t lub () w przeciwnym wypadku.

```
(defun and. (x y)
  (cond (x (cond (y 't) ('t '())))
        ('t '())))
> (and. (atom 'a) (eq 'a 'a))
t
> (and. (atom 'a) (eq 'a 'b))
()
```

Operator logiczny NOT

③ `(not. x)` zwraca wartość przeciwna argumentu x.

```
(defun not. (x)
  (cond (x '())
        ('t 't)))
> (not (eq 'a 'a))
()
> (not (eq 'a 'b))
t
```

Sklejanie dwóch list (konkatenacja)

④ (append. x y) skleja dwie listy x i y.

```
(defun append. (x y)
  (cond ((null. x) y)
        ('t (cons car x) (append. (cdr x) y)))))
> (append. '(a b) '(c d))
(a b c d)
> (append. '() '(c d))
(c d)
```

Łaczenie elementów list w pary (listy asocjacyjne)

⑤ (pair. x y) pobiera 2 listy tej samej długości i zwraca liste dwuelementowych podlist z których każda składa sie z kolejnych elementów x i odpowiadajacych im na tej samej pozycji elementów listy y.

```
(defun pair. (x y)
  (cond ((and. (null. x) (null. y)) '())
        ((and. (not. (atom x)) (not. (atom y)))
        (cons (list (car x) (car y))
              (pair. (cdr x) (cdr y))))))
> (pair. '(x y z) '(a b c))
((x a) (y b) (z c))
```

Pobranie wartości etykiety z listy asocjacyjnej

⑥ `(assoc.` x y`)` pobiera atom x oraz liste y postaci zwróconej przez `pair.` oraz zwraca drugi element w pierwszej podliście której pierwszy element odpowiada x.

```
(defun assoc. (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc. x (cdr y)))))
> (assoc. 'x '((x a) (y b)))
a
> (assoc. 'x '((x new) (x a) (y b)))
new
```

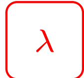
Tworzenie interpretera za pomoca funkcji

⑦ (evcon. c a) pomocnicza funkcja dla przetwarzania instrukcji warunkowych.

```
(defun evcon. (c a)
  (cond ((eval. (caar c) a)
         (eval. (cadar c) a))
        ('t (evcon. (cdr c) a))))
```

⑧ (evlis. m a) pomocnicza funkcja dla przetwarzania list.

```
(defun evlis. (m a)
  (cond ((null. m) '())
        ('t (cons (eval.  (car m) a)
                  (evlis. (cdr m) a)))))
```



Definiowanie funkcji – interpretera

```
(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom)  (atom   (eval. (cadr e) a)))
       ((eq (car e) 'eq)    (eq     (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'car)   (car    (eval. (cadr e) a)))
       ((eq (car e) 'cdr)   (cdr    (eval. (cadr e) a)))
       ((eq (car e) 'cons)  (cons   (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'cond)  (evcon. (cdr e) a))
   ...
```

Definiowanie funkcji – interpretera (c.d.)

```
    ...
       ('t (eval. (cons (assoc. (car e) a)
                        (cdr e))
                  a))))
    ((eq (caar e) 'label)
     (eval. (cons (caddar e) (cdr e))
            (cons (list. (cadar e) (car e)) a)))
    ((eq (caar e) 'lambda)
     (eval. (caddar e)
            (append. (pair. (cadar e) (evlis. (cdr e) a))
                     a)))))
```

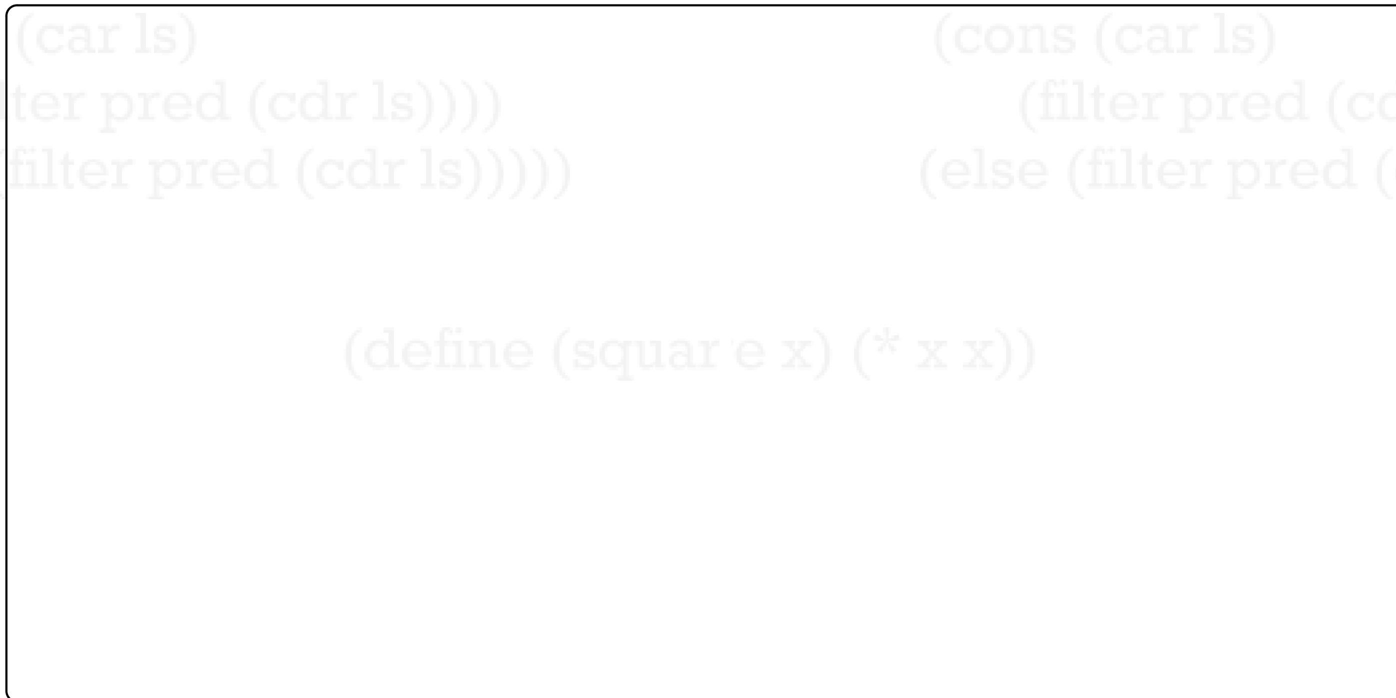
Definicje rekurencyjne

Przykładowa definicja rekurencyjna funkcji `silnia` w jezyku Lisp

```
(defun silnia (n)
  (cond ((= 0 n) 1)
        (t (* n (silnia (- n 1))))))
> (silnia 32)
263130836933693530167218012160000000
> (silnia 4)
> 24
> (silnia 1024)
541852879605885728307692194468385473800155396353801347068321061
207337660373314098413621458671907918845708994165770187368260454
133333721939108367528012764993769937891165755680659663747947314
518404886677672556125181213677274521963430770133713205796248433
128870088436137518390452944732277808402932158722061853806162806
063822186848239287130261690914211362251144684713888587884046...
```

Definicje rekurencyjne c.d.

Etapy obliczeń rekurencyjnych na przykładzie funkcji obliczajacej silnie



Definicje rekurencyjne c.d.

Etapy obliczeń rekurencyjnych na przykładzie funkcji obliczajacej silnie

```
❶ (silnia 4)
```

(file-listing dir "*")) (if (eq? n 0) #f (:file-listing dir "*")) (if (eq? n 0) #f
(cond ((null? ls) '()) ((pred (car ls)) (cons (car ls) (filter pred (cdr ls)))) (else (filter pred (cdr ls)))))
(define (square x) (* x x))

Definicje rekurencyjne c.d.

Etapy obliczeń rekurencyjnych na przykładzie funkcji obliczajacej silnie

```
❶ (silnia 4)
❷ (* 4 (silnia 3))
```

Definicje rekurencyjne c.d.

Etapy obliczeń rekurencyjnych na przykładzie funkcji obliczajacej silnie

```
❶ (silnia 4)
❷ (* 4 (silnia 3))
❸ (* 4 (* 3 (silnia 2)))
```

file−listing dir "*")) (if (eq? n 0) #f (:file−listing dir "*")) (if (eq? n 0) #f
(cond ((null? ls) '()) ((pred (car ls)) (cons (car ls) (filter pred (cdr ls))) (else (filter pred (cdr ls)))))
e x) (* x x)) (define (squar e x) (* x x)) (define (squa

Definicje rekurencyjne c.d.

Etapy obliczeń rekurencyjnych na przykładzie funkcji obliczajacej silnie

```
❶ (silnia 4)
❷ (* 4 (silnia 3))
❸ (* 4 (* 3 (silnia 2)))
❹ (* 4 (* 3 (* 2 (silnia 1))))
```

Definicje rekurencyjne c.d.

Etapy obliczeń rekurencyjnych na przykładzie funkcji obliczajacej silnie

```
❶ (silnia 4)
❷ (* 4 (silnia 3))
❸ (* 4 (* 3 (silnia 2)))
❹ (* 4 (* 3 (* 2 (silnia 1))))
❺ (* 4 (* 3 (* 2 (* 1 (silnia 0)))))
```

Definicje rekurencyjne c.d.

Etapy obliczeń rekurencyjnych na przykładzie funkcji obliczajacej silnie

```
❶ (silnia 4)
❷ (* 4 (silnia 3))
❸ (* 4 (* 3 (silnia 2)))
❹ (* 4 (* 3 (* 2 (silnia 1))))
❺ (* 4 (* 3 (* 2 (* 1 (silnia 0)))))
❺ (* 4 (* 3 (* 2 (* 1 1)))) ; <== 1
```

Definicje rekurencyjne c.d.

Etapy obliczeń rekurencyjnych na przykładzie funkcji obliczajacej silnie

```
❶ (silnia 4)
❷ (* 4 (silnia 3))
❸ (* 4 (* 3 (silnia 2)))
❹ (* 4 (* 3 (* 2 (silnia 1))))
❺ (* 4 (* 3 (* 2 (* 1 (silnia 0)))))
❺ (* 4 (* 3 (* 2 (* 1 1))))
❹ (* 4 (* 3 (* 2 1))) ; <== 1
```

Definicje rekurencyjne c.d.

Etapy obliczeń rekurencyjnych na przykładzie funkcji obliczajacej silnie

```
❶ (silnia 4)
❷ (* 4 (silnia 3))
❸ (* 4 (* 3 (silnia 2)))
❹ (* 4 (* 3 (* 2 (silnia 1))))
❺ (* 4 (* 3 (* 2 (* 1 (silnia 0)))))
❺ (* 4 (* 3 (* 2 (* 1 1))))
❹ (* 4 (* 3 (* 2 1)))
❸ (* 4 (* 3 2)) ; <== 2
```

Definicje rekurencyjne c.d.

Etapy obliczeń rekurencyjnych na przykładzie funkcji obliczajacej silnie

```
❶ (silnia 4)
❷ (* 4 (silnia 3))
❸ (* 4 (* 3 (silnia 2)))
❹ (* 4 (* 3 (* 2 (silnia 1))))
❺ (* 4 (* 3 (* 2 (* 1 (silnia 0)))))
❺ (* 4 (* 3 (* 2 (* 1 1))))
❹ (* 4 (* 3 (* 2 1)))
❸ (* 4 (* 3 2))
❷ (* 4 6) ; <== 6
```

Definicje rekurencyjne c.d.

Etapy obliczeń rekurencyjnych na przykładzie funkcji obliczajacej silnie

```
❶ (silnia 4)
❷ (* 4 (silnia 3))
❸ (* 4 (* 3 (silnia 2)))
❹ (* 4 (* 3 (* 2 (silnia 1))))
❺ (* 4 (* 3 (* 2 (* 1 (silnia 0)))))
❺ (* 4 (* 3 (* 2 (* 1 1))))
❹ (* 4 (* 3 (* 2 1)))
❸ (* 4 (* 3 2))
❷ (* 4 6)
❶ 24 ; <== 24
```

SCHEME: Elementarne operacje na listach (powtórka)

`car` Funkcja zwracajaca pierwszy element listy lub pary. Zwraca zawsze atom badź liste (w niektórych implementacjach wystepuje także pod nazwa `first`).

`cdr` Funkcja zwracajaca liste podana jako argument wywołania za wyjatkiem pierwszego elementu. Zawsze zwraca liste. (W przypadku pary zwraca jej drugi element). (w niektórych implementacjach wystepuje także pod nazwa `rest`).

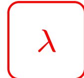
`cons` Funkcja dodaje element e na poczatek listy l.
e – `atom | lista`
l – `lista`

SCHEME: Elementarne operacje na listach (powtórka)

`car` Funkcja zwracajaca pierwszy element listy lub pary. Zwraca zawsze atom badź liste (w niektórych implementacjach wystepuje także pod nazwa `first`).

`cdr` Funkcja zwracajaca liste podana jako argument wywołania za wyjatkiem pierwszego elementu. Zawsze zwraca liste. (W przypadku pary zwraca jej drugi element). (w niektórych implementacjach wystepuje także pod nazwa `rest`).

`cons` Funkcja dodaje element e na poczatek listy l.
e – `atom | lista`
l – `lista`



SCHEME: Elementarne operacje na listach (powtórka)

`car` Funkcja zwracajaca pierwszy element listy lub pary. Zwraca zawsze atom badź liste (w niektórych implementacjach wystepuje także pod nazwa `first`).

`cdr` Funkcja zwracajaca liste podana jako argument wywołania za wyjatkiem pierwszego elementu. Zawsze zwraca liste. (W przypadku pary zwraca jej drugi element). (w niektórych implementacjach wystepuje także pod nazwa `rest`).

`cons` Funkcja dodaje element e na poczatek listy l.
e – `atom | lista`
l – `lista`

SCHEME: Typy danych i ich określanie

- ✓ **Predykaty.** Konwencja nazewnicza: funkcje kończace sie znakiem ? zwracaja wartości `#t|#f`. Służa np. do sprawdzania rodzaju argumentów wewnatrz funkcji (brak formalnej definicji typów argumentów w definiowanych funkcjach). Intuicyjne nazewnictwo: `boolean?`, `number?`, `pair?`, `symbol?`, `procedure?`, `null?`, `zero?`, `odd?`, `even?` . . .

- ✓ **Operatory logiczne.** `and`, `or`, `not`.

- ✓ **Operatory relacji.** =, <, <=, >, >=.

- ✓ **Porównywanie argumentów.**

  - `eq?` – `#t` jeśli sa identyczne
  - `eqv?` – `#t` jeśli sa ekwiwalentne operacyjnie
  - `equal?` – `#t` jeśli maja ta sama strukture i zawartość

SCHEME: Typy danych i ich określanie

- ✓ **Predykaty.** Konwencja nazewnicza: funkcje kończace sie znakiem ? zwracaja wartości #t|#f. Służa np. do sprawdzania rodzaju argumentów wewnatrz funkcji (brak formalnej definicji typów argumentów w definiowanych funkcjach). Intuicyjne nazewnictwo: `boolean?`, `number?`, `pair?`, `symbol?`, `procedure?`, `null?`, `zero?`, `odd?`, `even?` . . .

- ✓ **Operatory logiczne.** `and`, `or`, `not`.

- ✓ **Operatory relacji.** =, <, <=, >, >=.

- ✓ **Porównywanie argumentów.**

  – `eq?` – `#t` jeśli sa identyczne
  – `eqv?` – `#t` jeśli sa ekwiwalentne operacyjnie
  – `equal?` – `#t` jeśli maja ta sama strukture i zawartość

SCHEME: Typy danych i ich określanie

- ✓ **Predykaty.** Konwencja nazewnicza: funkcje kończace sie znakiem ? zwracaja wartości `#t|#f`. Służa np. do sprawdzania rodzaju argumentów wewnatrz funkcji (brak formalnej definicji typów argumentów w definiowanych funkcjach). Intuicyjne nazewnictwo: `boolean?`, `number?`, `pair?`, `symbol?`, `procedure?`, `null?`, `zero?`, `odd?`, `even?` . . .

- ✓ **Operatory logiczne.** `and`, `or`, `not`.

- ✓ **Operatory relacji.** =, <, <=, >, >=.

- ✓ **Porównywanie argumentów.**

  - eq? – `#t` jeśli sa identyczne
  - eqv? – `#t` jeśli sa ekwiwalentne operacyjnie
  - `equal?` – `#t` jeśli maja ta sama strukture i zawartość

SCHEME: Typy danych i ich określanie

✓ **Predykaty.** Konwencja nazewnicza: funkcje kończace sie znakiem ? zwracaja wartości `#t|#f`. Służa np. do sprawdzania rodzaju argumentów wewnatrz funkcji (brak formalnej definicji typów argumentów w definiowanych funkcjach). Intuicyjne nazewnictwo: `boolean?`, `number?`, `pair?`, `symbol?`, `procedure?`, `null?`, `zero?`, `odd?`, `even?` . . .

✓ **Operatory logiczne.** `and`, `or`, `not`.

✓ **Operatory relacji.** =, <, <=, >, >=.

✓ **Porównywanie argumentów.**

– `eq?` – `#t` jeśli sa identyczne
– `eqv?` – `#t` jeśli sa ekwiwalentne operacyjnie
– `equal?` – `#t` jeśli maja ta sama strukture i zawartość

SCHEME: Typy danych i ich określanie

- ✓ **Predykaty.** Konwencja nazewnicza: funkcje kończace sie znakiem ? zwracaja wartości `#t|#f`. Służa np. do sprawdzania rodzaju argumentów wewnatrz funkcji (brak formalnej definicji typów argumentów w definiowanych funkcjach). Intuicyjne nazewnictwo: `boolean?`, `number?`, `pair?`, `symbol?`, `procedure?`, `null?`, `zero?`, `odd?`, `even?` . . .

- ✓ **Operatory logiczne.** `and`, `or`, `not`.

- ✓ **Operatory relacji.** =, <, <=, >, >=.

- ✓ **Porównywanie argumentów.**

  – eq? – `#t` jeśli sa identyczne
  – eqv? – `#t` jeśli sa ekwiwalentne operacyjnie
  – `equal?` – `#t` jeśli maja ta sama strukture i zawartość

SCHEME: Typy danych i ich określanie

✓ **Predykaty.** Konwencja nazewnicza: funkcje kończace sie znakiem ? zwracaja wartości #t|#f. Służa np. do sprawdzania rodzaju argumentów wewnatrz funkcji (brak formalnej definicji typów argumentów w definiowanych funkcjach). Intuicyjne nazewnictwo: boolean?, number?, pair?, symbol?, procedure?, null?, zero?, odd?, even? . . .

✓ **Operatory logiczne.** and, or, not.

✓ **Operatory relacji.** =, <, <=, >, >=.

✓ **Porównywanie argumentów.**

- eq? – #t jeśli sa identyczne
- eqv? – #t jeśli sa ekwiwalentne operacyjnie
- equal? – #t jeśli maja ta sama strukture i zawartość

SCHEME: Typy danych i ich określanie

- ✓ **Predykaty.** Konwencja nazewnicza: funkcje kończace sie znakiem ? zwracaja wartości `#t|#f`. Służa np. do sprawdzania rodzaju argumentów wewnatrz funkcji (brak formalnej definicji typów argumentów w definiowanych funkcjach). Intuicyjne nazewnictwo: `boolean?`, `number?`, `pair?`, `symbol?`, `procedure?`, `null?`, `zero?`, `odd?`, `even?` . . .

- ✓ **Operatory logiczne.** `and`, `or`, `not`.

- ✓ **Operatory relacji.** =, <, <=, >, >=.

- ✓ **Porównywanie argumentów.**

  - eq? – `#t` jeśli sa identyczne
  - eqv? – `#t` jeśli sa ekwiwalentne operacyjnie
  - `equal?` – `#t` jeśli maja ta sama strukture i zawartość

SCHEME: Instrukcje warunkowe

❶ **if**
`(if (`p`) `e_y` `e_n`)` Jeśli p jest spełnione (zwraca `#t`) zostanie wykonane e_y, w przeciwnym wypadku e_n.

❷ **cond**
`(cond (`p_1` `e_1`) ... (`p_n` `e_n`) (else `e_e`)` Kolejne wyrażenia p podlegaja przetworzeniu do momentu, gdy któreś z nich zwróci wartość `#t` (wtedy przetworzeniu ulegnie odpowiednie wyrażenie e). Jeśli żadne z p nie zwróci `#t` przetworzeniu ulegnie e_e.

SCHEME: Instrukcje warunkowe

❶ **if**

`(if` (p) e_y e_n`)` Jeśli p jest spełnione (zwraca `#t`) zostanie wykonane e_y, w przeciwnym wypadku e_n.

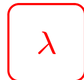
❷ **cond**

`(cond` $(p_1\ \ e_1)$ `...` $(p_n\ \ e_n)$ `(else` e_e`)` Kolejne wyrażenia p podlegaja przetworzeniu do momentu, gdy któreś z nich zwróci wartość `#t` (wtedy przetworzeniu ulegnie odpowiednie wyrażenie e). Jeśli żadne z p nie zwróci `#t` przetworzeniu ulegnie e_e.

✓ **Funkcje przekształceń.** (*mutation procedures*). Konwencja nazewnicza: funkcje kończace sie znakiem ! modyfikuja wartości swoich argumentów. Służa np. do zmiany wartości zmiennych: `set!`.

✓ **Deklarowanie funkcji i zmiennych.**

- Przypisanie wartości c do zmiennej v: `(define` v c`)`
- Definiowanie funkcji f przyjmujacej liste a_n argumentów:
  `(define (`f a_1 a_2 ... a_n`)`
  `(`$body$`))`
- Definiowanie funkcji f przy pomocy `lambda`:
  `(define` f `(lambda (`a_1 a_2 ... a_n`) (`$body$`)))`
- Deklarowanie zmiennych lokalnych:
  `(let (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`
- Deklarowanie zmiennych lokalnych (przypisywanie w kolejności):
  `(let* (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`
- Deklarowanie zmiennych lokalnych dla procedur rekurencyjnych:
  `(letrec (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`



✓ **Funkcje przekształceń.** (*mutation procedures*). Konwencja nazewnicza: funkcje kończace sie znakiem `!` modyfikuja wartości swoich argumentów. Służa np. do zmiany wartości zmiennych: `set!`.

✓ **Deklarowanie funkcji i zmiennych.**

– Przypisanie wartości c do zmiennej v: `(define` v c`)`
– Definiowanie funkcji f przyjmujacej liste a_n argumentów:
`(define (`f a_1 a_2 ... a_n`)`
`(`$body$`))`
– Definiowanie funkcji f przy pomocy `lambda`:
`(define` f `(lambda (`a_1 a_2 ... a_n`) (`$body$`)))`
– Deklarowanie zmiennych lokalnych:
`(let (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`
– Deklarowanie zmiennych lokalnych (przypisywanie w kolejności):
`(let* (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`
– Deklarowanie zmiennych lokalnych dla procedur rekurencyjnych:
`(letrec (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`

✓ **Funkcje przekształceń.** (*mutation procedures*). Konwencja nazewnicza: funkcje kończace sie znakiem `!` modyfikuja wartości swoich argumentów. Służa np. do zmiany wartości zmiennych: `set!`.

✓ **Deklarowanie funkcji i zmiennych.**

- Przypisanie wartości c do zmiennej v: `(define` v c`)`
- Definiowanie funkcji f przyjmujacej liste a_n argumentów:
  `(define (`f a_1 a_2 ... a_n`)`
  `(`$body$`))`
- Definiowanie funkcji f przy pomocy `lambda`:
  `(define` f `(lambda (`a_1 a_2 ... a_n`) (`$body$`)))`
- Deklarowanie zmiennych lokalnych:
  `(let (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`
- Deklarowanie zmiennych lokalnych (przypisywanie w kolejności):
  `(let* (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`
- Deklarowanie zmiennych lokalnych dla procedur rekurencyjnych:
  `(letrec (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`

- ✓ **Funkcje przekształceń.** (*mutation procedures*). Konwencja nazewnicza: funkcje kończace sie znakiem ! modyfikuja wartości swoich argumentów. Służa np. do zmiany wartości zmiennych: `set!`.

- ✓ **Deklarowanie funkcji i zmiennych.**
  - Przypisanie wartości c do zmiennej v: `(define` v c`)`
  - Definiowanie funkcji f przyjmujacej liste a_n argumentów: `(define (`f a_1 a_2 ... a_n`)` `(`$body$`))`
  - Definiowanie funkcji f przy pomocy `lambda`: `(define` f `(lambda (`a_1 a_2 ... a_n`)` `(`$body$`)))`
  - Deklarowanie zmiennych lokalnych: `(let (`v_1 c_1`) ... (`v_n c_n`)` `(`$body$`))`
  - Deklarowanie zmiennych lokalnych (przypisywanie w kolejności): `(let* (`v_1 c_1`) ... (`v_n c_n`)` `(`$body$`))`
  - Deklarowanie zmiennych lokalnych dla procedur rekurencyjnych: `(letrec (`v_1 c_1`) ... (`v_n c_n`)` `(`$body$`))`

✓ **Funkcje przekształceń.** (*mutation procedures*). Konwencja nazewnicza: funkcje kończace sie znakiem ! modyfikuja wartości swoich argumentów. Służa np. do zmiany wartości zmiennych: `set!`.

✓ **Deklarowanie funkcji i zmiennych.**

– Przypisanie wartości c do zmiennej v: `(define` v c`)`
– Definiowanie funkcji f przyjmujacej liste a_n argumentów:
`(define (`f a_1 a_2 ... a_n`)`
`(`$body$`))`
– Definiowanie funkcji f przy pomocy `lambda`:
`(define` f `(lambda (`a_1 a_2 ... a_n`) (`$body$`)))`
– Deklarowanie zmiennych lokalnych:
`(let (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`
– Deklarowanie zmiennych lokalnych (przypisywanie w kolejności):
`(let* (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`
– Deklarowanie zmiennych lokalnych dla procedur rekurencyjnych:
`(letrec (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`

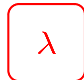
✓ **Funkcje przekształceń.** (*mutation procedures*). Konwencja nazewnicza: funkcje kończace sie znakiem `!` modyfikuja wartości swoich argumentów. Służa np. do zmiany wartości zmiennych: `set!`.

✓ **Deklarowanie funkcji i zmiennych.**

– Przypisanie wartości c do zmiennej v: (`define` v c)
– Definiowanie funkcji f przyjmujacej liste a_n argumentów:
(`define` (f a_1 a_2 ... a_n)
($body$))
– Definiowanie funkcji f przy pomocy `lambda`:
(`define` f (`lambda` (a_1 a_2 ... a_n) ($body$)))
– Deklarowanie zmiennych lokalnych:
(`let` (v_1 c_1) ... (v_n c_n) ($body$))
– Deklarowanie zmiennych lokalnych (przypisywanie w kolejności):
(`let*` (v_1 c_1) ... (v_n c_n) ($body$))
– Deklarowanie zmiennych lokalnych dla procedur rekurencyjnych:
(`letrec` (v_1 c_1) ... (v_n c_n) ($body$))

- ✓ **Funkcje przekształceń.** (*mutation procedures*). Konwencja nazewnicza: funkcje kończace sie znakiem `!` modyfikuja wartości swoich argumentów. Służa np. do zmiany wartości zmiennych: `set!`.

- ✓ **Deklarowanie funkcji i zmiennych.**
  - Przypisanie wartości c do zmiennej v: `(define` v c`)`
  - Definiowanie funkcji f przyjmujacej liste a_n argumentów:
    `(define (`f a_1 a_2 … a_n`)`
    `(`$body$`))`
  - Definiowanie funkcji f przy pomocy `lambda`:
    `(define` f `(lambda (`a_1 a_2 … a_n`) (`$body$`)))`
  - Deklarowanie zmiennych lokalnych:
    `(let (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`
  - Deklarowanie zmiennych lokalnych (przypisywanie w kolejności):
    `(let* (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`
  - Deklarowanie zmiennych lokalnych dla procedur rekurencyjnych:
    `(letrec (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`



✓ **Funkcje przekształceń.** (*mutation procedures*). Konwencja nazewnicza: funkcje kończace sie znakiem `!` modyfikuja wartości swoich argumentów. Służa np. do zmiany wartości zmiennych: `set!`.

✓ **Deklarowanie funkcji i zmiennych.**

- Przypisanie wartości c do zmiennej v: `(define` v c`)`
- Definiowanie funkcji f przyjmujacej liste a_n argumentów:
  `(define (`f a_1 a_2 ... a_n`)`
  `(`$body$`))`
- Definiowanie funkcji f przy pomocy `lambda`:
  `(define` f `(lambda (`a_1 a_2 ... a_n`) (`$body$`)))`
- Deklarowanie zmiennych lokalnych:
  `(let (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`
- Deklarowanie zmiennych lokalnych (przypisywanie w kolejności):
  `(let* (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`
- Deklarowanie zmiennych lokalnych dla procedur rekurencyjnych:
  `(letrec (`v_1 c_1`) ... (`v_n c_n`) (`$body$`))`

Zastosowania jezyków Lisp/Scheme

✓ Dominujacy od poczatków powstania w aplikacjach AI, w szczególności:

– AI (robotyka, gry komputerowe [od Craps, BlackJack do Age of Empires], rozpoznawanie wzorców, chatter boty /test Turinga/ od Elizy do bardziej współczesnych i rozbudowanych:
`http://congsci.ucsd.edu/ asaygin/tt/ttest.html`

– Systemy obrony powietrznej.

– Osadzone systemy 'manipulacji wiedza' (także RT), przetwarzania listowego i algorytmów grafowych.

– Skalowalny jezyk skryptowy, jezyk rozszerzeń (Emacs Lisp/GUILE szereg innych osadzanych interpreterów).

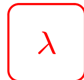
– System ACT-R (modele architektur kognitywnych).

– Szereg innych mniej popularnych zastosowań. . .

Zastosowania jezyków Lisp/Scheme

✓ Dominujacy od poczatków powstania w aplikacjach AI, w szczególności:

- AI (robotyka, gry komputerowe [od Craps, BlackJack do Age of Empires], rozpoznawanie wzorców, chatter boty /test Turinga/ od Elizy do bardziej współczesnych i rozbudowanych:
  `http://congsci.ucsd.edu/ asaygin/tt/ttest.html`
- Systemy obrony powietrznej.
- Osadzone systemy 'manipulacji wiedza' (także RT), przetwarzania listowego i algorytmów grafowych.
- Skalowalny jezyk skryptowy, jezyk rozszerzeń (Emacs Lisp/GUILE szereg innych osadzanych interpreterów).
- System ACT-R (modele architektur kognitywnych).
- Szereg innych mniej popularnych zastosowań. . .



Zastosowania jezyków Lisp/Scheme

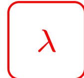
✓ Dominujacy od poczatków powstania w aplikacjach AI, w szczególności:

- AI (robotyka, gry komputerowe [od Craps, BlackJack do Age of Empires], rozpoznawanie wzorców, chatter boty /test Turinga/ od Elizy do bardziej współczesnych i rozbudowanych: `http://congsci.ucsd.edu/ asaygin/tt/ttest.html`
- Systemy obrony powietrznej.
- Osadzone systemy 'manipulacji wiedza' (także RT), przetwarzania listowego i algorytmów grafowych.
- Skalowalny jezyk skryptowy, jezyk rozszerzeń (Emacs Lisp/GUILE szereg innych osadzanych interpreterów).
- System ACT-R (modele architektur kognitywnych).
- Szereg innych mniej popularnych zastosowań. . .

Zastosowania jezyków Lisp/Scheme

✓ Dominujacy od poczatków powstania w aplikacjach AI, w szczególności:

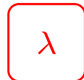
- AI (robotyka, gry komputerowe [od Craps, BlackJack do Age of Empires], rozpoznawanie wzorców, chatter boty /test Turinga/ od Elizy do bardziej współczesnych i rozbudowanych: `http://congsci.ucsd.edu/ asaygin/tt/ttest.html`
- Systemy obrony powietrznej.
- Osadzone systemy 'manipulacji wiedza' (także RT), przetwarzania listowego i algorytmów grafowych.
- Skalowalny jezyk skryptowy, jezyk rozszerzeń (Emacs Lisp/GUILE szereg innych osadzanych interpreterów).
- System ACT-R (modele architektur kognitywnych).
- Szereg innych mniej popularnych zastosowań. . .



Zastosowania jezyków Lisp/Scheme

✓ Dominujacy od poczatków powstania w aplikacjach AI, w szczególności:

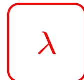
- AI (robotyka, gry komputerowe [od Craps, BlackJack do Age of Empires], rozpoznawanie wzorców, chatter boty /test Turinga/ od Elizy do bardziej współczesnych i rozbudowanych:
  `http://congsci.ucsd.edu/ asaygin/tt/ttest.html`
- Systemy obrony powietrznej.
- Osadzone systemy 'manipulacji wiedza' (także RT), przetwarzania listowego i algorytmów grafowych.
- Skalowalny jezyk skryptowy, jezyk rozszerzeń (Emacs Lisp/GUILE szereg innych osadzanych interpreterów).
- System ACT-R (modele architektur kognitywnych).
- Szereg innych mniej popularnych zastosowań. . .



Zastosowania jezyków Lisp/Scheme

✓ Dominujacy od poczatków powstania w aplikacjach AI, w szczególności:

- AI (robotyka, gry komputerowe [od Craps, BlackJack do Age of Empires], rozpoznawanie wzorców, chatter boty /test Turinga/ od Elizy do bardziej współczesnych i rozbudowanych:
  `http://congsci.ucsd.edu/ asaygin/tt/ttest.html`
- Systemy obrony powietrznej.
- Osadzone systemy 'manipulacji wiedza' (także RT), przetwarzania listowego i algorytmów grafowych.
- Skalowalny jezyk skryptowy, jezyk rozszerzeń (Emacs Lisp/GUILE szereg innych osadzanych interpreterów).
- System ACT-R (modele architektur kognitywnych).
- Szereg innych mniej popularnych zastosowań. . .



Zastosowania jezyków Lisp/Scheme

✓ Dominujacy od poczatków powstania w aplikacjach AI, w szczególności:

- AI (robotyka, gry komputerowe [od Craps, BlackJack do Age of Empires], rozpoznawanie wzorców, chatter boty /test Turinga/ od Elizy do bardziej współczesnych i rozbudowanych: `http://congsci.ucsd.edu/ asaygin/tt/ttest.html`
- Systemy obrony powietrznej.
- Osadzone systemy 'manipulacji wiedza' (także RT), przetwarzania listowego i algorytmów grafowych.
- Skalowalny jezyk skryptowy, jezyk rozszerzeń (Emacs Lisp/GUILE szereg innych osadzanych interpreterów).
- System ACT-R (modele architektur kognitywnych).
- Szereg innych mniej popularnych zastosowań. . .

The End

(if (eq? n 0) #f (:file-listing dir "*"))

(cond ((null? ls) '())
((pred (car ls))
(cons (car ls)
(filter pred (cdr ls))))
(else (filter pred (cdr ls)))))

(define (square x) (* x x))