

# Początki Lispu

## ((Archeologia Cyfrowa)))

Jakub Grobelny

19.11.2019

# O czym będzie?

- 1 Do czego był potrzebny taki język?
- 2 „*Recursive Functions of Symbolic Expressions...*”
- 3 Pierwsze implementacje
- 4 Być może coś więcej

# Narodziny sztucznej inteligencji

# Narodziny sztucznej inteligencji

Lata 50. XX wieku były czasem, gdy sztuczna inteligencja pojawiła się jako dziedzina wiedzy.

# Narodziny sztucznej inteligencji

Lata 50. XX wieku były czasem, gdy sztuczna inteligencja pojawiła się jako dziedzina wiedzy.

1950 Alan Turing publikuje „*Computing Machinery and Intelligence*”

# Narodziny sztucznej inteligencji

Lata 50. XX wieku były czasem, gdy sztuczna inteligencja pojawiła się jako dziedzina wiedzy.

1950 Alan Turing publikuje „*Computing Machinery and Intelligence*”

- „Czy maszyny myślą?”

# Narodziny sztucznej inteligencji

Lata 50. XX wieku były czasem, gdy sztuczna inteligencja pojawiła się jako dziedzina wiedzy.

1950 Alan Turing publikuje „*Computing Machinery and Intelligence*”

- „Czy maszyny myślą?”
- „Czy maszyny mogą myśleć?”

# Narodziny sztucznej inteligencji

Lata 50. XX wieku były czasem, gdy sztuczna inteligencja pojawiła się jako dziedzina wiedzy.

1950 Alan Turing publikuje „*Computing Machinery and Intelligence*”

- „~~Czy maszyny myślą?~~”
- „~~Czy maszyny mogą myśleć?~~”
- „Czy maszyny mogą działać nieodróżnialnie od ludzi?”



# Narodziny sztucznej inteligencji

Lata 50. XX wieku były czasem, gdy sztuczna inteligencja pojawiła się jako dziedzina wiedzy.

1950 Alan Turing publikuje „*Computing Machinery and Intelligence*”

- „~~Czy maszyny myślą?~~”
- „~~Czy maszyny mogą myśleć?~~”
- „Czy maszyny mogą działać nieodróżnialnie od ludzi?”

1951 Pierwsze programy grające w warcaby (Christopher Strachey) i szachy (Dietrich Prinz)

# Narodziny sztucznej inteligencji

1955 Allen Newell, Herbert A. Simon i Cliff Shaw tworzą program „*Logic Theorist*”

# Narodziny sztucznej inteligencji

1955 Allen Newell, Herbert A. Simon i Cliff Shaw tworzą program „*Logic Theorist*”

- Program naśladujący ludzkie techniki rozwiązywania problemów

# Narodziny sztucznej inteligencji

1955 Allen Newell, Herbert A. Simon i Cliff Shaw tworzą program „*Logic Theorist*”

- Program naśladowujący ludzkie techniki rozwiązywania problemów
- Program manipulujący **symbolami**

# Narodziny sztucznej inteligencji

1955 Allen Newell, Herbert A. Simon i Cliff Shaw tworzą program „*Logic Theorist*”

- Program naśladowujący ludzkie techniki rozwiązywania problemów
- Program manipulujący **symbolami**
- Udowodnił 38 z pierwszych 52 twierdzeń z „*Principia Mathematica*”<sup>1</sup>

---

<sup>2</sup>Autorstwa Alfreda Northa Whiteheada i Bertranda Russella

# Narodziny sztucznej inteligencji

1955 Allen Newell, Herbert A. Simon i Cliff Shaw tworzą program „*Logic Theorist*”

- Program naśladowujący ludzkie techniki rozwiązywania problemów
- Program manipulujący **symbolami**
- Udowodnił 38 z pierwszych 52 twierdzeń z „*Principia Mathematica*”<sup>1</sup> (niektóre z dowodów były bardziej eleganckie niż wcześniej istniejące)

---

<sup>2</sup>Autorstwa Alfreda Northa Whiteheada i Bertranda Russella

# Narodziny sztucznej inteligencji

1955 Allen Newell, Herbert A. Simon i Cliff Shaw tworzą program „*Logic Theorist*”

- Program naśladowujący ludzkie techniki rozwiązywania problemów
- Program manipulujący **symbolami**
- Udowodnił 38 z pierwszych 52 twierdzeń z „*Principia Mathematica*”<sup>1</sup> (niektóre z dowodów były bardziej eleganckie niż wcześniej istniejące)

# Narodziny sztucznej inteligencji

1956 Konferencja w  
Dartmouth.



# Narodziny sztucznej inteligencji

1956 Konferencja w  
Dartmouth. **John  
McCarthy** przekonuje  
zebranych do używania  
terminu „*sztuczna  
inteligencja*”.



Rysunek: John McCarthy

# Symbole

# Symbole

Okazuje się, że operowanie na **symbolach** jest istotne przy tworzeniu sztucznej inteligencji.

# Symbole

Okazuje się, że operowanie na **symbolach** jest istotne przy tworzeniu sztucznej inteligencji.

Ludzkie rozumowanie opiera się na manipulacji symbolami (*physical symbol system hypothesis* – Allan Newell i Herbert A. Simon)

# Symbole

Okazuje się, że operowanie na **symbolach** jest istotne przy tworzeniu sztucznej inteligencji.

Ludzkie rozumowanie opiera się na manipulacji symbolami (*physical symbol system hypothesis* – Allan Newell i Herbert A. Simon)

System symboli składa się z symboli, składania ich w struktury (wyrażenia) i manipulowania nimi (przetwarzania) w celu tworzenia nowych wyrażień.

# Symbole

W 1959 roku John McCarthy pisze pracę „*Programs with common sense*” zainspirowaną programem „Logic Theorist”.

# Symbole

W 1959 roku John McCarthy pisze pracę „*Programs with common sense*” zainspirowaną programem „Logic Theorist”.

Proponuje w niej stworzenie programu „*Advice Taker*”, który rozwiązywałby problemy poprzez manipulację zdaniami (**symbole!**).

# Symbole

W 1959 roku John McCarthy pisze pracę „*Programs with common sense*” zainspirowaną programem „Logic Theorist”.

Proponuje w niej stworzenie programu „*Advice Taker*”, który rozwiązywałby problemy poprzez manipulację zdaniami (**symbole!**).

*„Our ultimate objective is to make programs that learn from their experience as effectively as humans do.”*



*„Programs with common sense”*

## „Programs with common sense”

*„A class of entities called terms is defined and a term is an expression. A sequence of expressions is an expression. These expressions are represented in the machine by list structures”*

## „Programs with common sense”

*„A class of entities called terms is defined and a term is an expression. A sequence of expressions is an expression. These expressions are represented in the machine by list structures”* – reprezentacja przesłanek/faktów w postaci **list symboli**

Przykłady:

*at(I, desk)*

*at(desk, home)*

*at(car, home)*

*at(home, county)*

*at(airport, county)*

Przykłady:

*at(I, desk)*

*at(desk, home)*

*at(car, home)*

*at(home, county)*

*at(airport, county)*

$at(x, y), at(y, z) \rightarrow at(x, z)$

Przykłady:

$at(I, desk)$

$at(desk, home)$

$at(car, home)$

$at(home, county)$

$at(airport, county)$

$at(x, y), at(y, z) \rightarrow at(x, z)$

$transitive(at)$

$transitive(u) \rightarrow (u(x, y), u(y, z) \rightarrow u(x, z))$

Przykłady:

*at(I, desk)*

*at(desk, home)*

*at(car, home)*

*at(home, county)*

*at(airport, county)*

$at(x, y), at(y, z) \rightarrow at(x, z)$

*transitive(at)*

$transitive(u) \rightarrow (u(x, y), u(y, z) \rightarrow u(x, z))$

Prawie jak Prolog?

# Zapotrzebowanie na nowe języki

**Information Processing Language** – niskopozimowy język programowania do manipulowania listami stworzony przez Newella, Shawa i Simona, który posłużył do napisania programu. „*Logic Theorist*”.



# Zapotrzebowanie na nowe języki

**Information Processing Language** – niskopozimowy język programowania do manipulowania listami stworzony przez Newella, Shawa i Simona, który posłużył do napisania programu. „*Logic Theorist*”.

- dane reprezentowane jako listy
- procedury operujące na danych

IPL-V List  
Structure  
Example

Name	SYMB	LINK
L1	9-1	100
100	S4	101
101	S5	0
9-1	0	200
200	A1	201
201	V1	202
202	A2	203
203	V2	0

Rysunek: Lista zapisana w IPL-V

# Information Processing Language

Nowe feature'y:

# Information Processing Language

Nowe feature'y:

- Manipulacja listami (tylko listy atomów)

# Information Processing Language

Nowe feature'y:

- Manipulacja listami (tylko listy atomów)
- Funkcje wyższego rzędu

# Information Processing Language

Nowe feature'y:

- Manipulacja listami (tylko listy atomów)
- Funkcje wyższego rzędu
- Obliczenia na symbolach (tylko litera+liczba)

# LISt Processor

# LISSt Processor

W 1960 ukazuje się praca Johna McCarthy'ego pt., *Recursive Functions of Symbolic Expressions Their Computation by Machine, Part I*".

W 1960 ukazuje się praca Johna McCarthy'ego pt., *Recursive Functions of Symbolic Expressions Their Computation by Machine, Part I*".

## 1. Introduction

A programming system called LISP (for LISSt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions.



# Wyrażenia warunkowe

Po raz pierwszy pojawiła się idea *wyrażeń warunkowych*.

# Wyrażenia warunkowe

Po raz pierwszy pojawiła się idea *wyrażeń warunkowych*.

We shall need a number of mathematical ideas and notations concerning functions in general. Most of the ideas are well known, but the notion of *conditional expression* is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.

# Wyrażenia warunkowe

Wyrażenia warunkowe miały mieć następującą postać:

$$(p_1 \rightarrow e_1, \dots, p_n \rightarrow e_n)$$

Interpretacja:

# Wyrażenia warunkowe

Wyrażenia warunkowe miały mieć następującą postać:

$$(p_1 \rightarrow e_1, \dots, p_n \rightarrow e_n)$$

Interpretacja:

„Jeżeli  $p_1$  to  $e_1$ , w przeciwnym razie jeżeli  $p_2$  to  $e_2$ , ... , w przeciwnym razie jeżeli  $p_n$  to  $e_n$ ”

# Wyrażenia warunkowe

Zasady obliczania wartości wyrażeń warunkowych:

# Wyrażenia warunkowe

Zasady obliczania wartości wyrażeń warunkowych:

- Rozpatruj  $p$  od lewej do prawej.

# Wyrażenia warunkowe

Zasady obliczania wartości wyrażeń warunkowych:

- Rozpatruj  $p$  od lewej do prawej.
- Jeżeli  $p$ , którego wartość to  $T$ , występuje przed jakimkolwiek innym  $p$ , którego wartość jest niezdefiniowana, to wartością wyrażenia warunkowego jest wartość odpowiadającego  $e$  (jeżeli jest zdefiniowane).

# Wyrażenia warunkowe

Zasady obliczania wartości wyrażeń warunkowych:

- Rozpatruj  $p$  od lewej do prawej.
- Jeżeli  $p$ , którego wartość to  $T$ , występuje przed jakimkolwiek innym  $p$ , którego wartość jest niezdefiniowana, to wartością wyrażenia warunkowego jest wartość odpowiadającego  $e$  (jeżeli jest zdefiniowane).
- Jeżeli jakiegokolwiek niezdefiniowane  $p$  jest napotkane przed prawdziwym  $p$ , lub gdy wszystkie  $p$  są fałszywe, bądź gdy  $e$  odpowiadającego pierwszemu prawdziwemu  $p$  jest niezdefiniowane, to wartość wyrażenia jest niezdefiniowana.



# Wyrażenia warunkowe

Przykłady:

# Wyrażenia warunkowe

Przykłady:

$$(2 < 1 \rightarrow 4, T \rightarrow 3) =$$

# Wyrażenia warunkowe

Przykłady:

$$(2 < 1 \rightarrow 4, T \rightarrow 3) = 3$$

# Wyrażenia warunkowe

Przykłady:

$$(2 < 1 \rightarrow 4, T \rightarrow 3) = 3$$

$$(2 < 1 \rightarrow \frac{0}{0}, T \rightarrow 3) =$$

# Wyrażenia warunkowe

Przykłady:

$$(2 < 1 \rightarrow 4, T \rightarrow 3) = 3$$

$$(2 < 1 \rightarrow \frac{0}{0}, T \rightarrow 3) = 3$$

# Wyrażenia warunkowe

Przykłady:

$$(2 < 1 \rightarrow 4, T \rightarrow 3) = 3$$

$$(2 < 1 \rightarrow \frac{0}{0}, T \rightarrow 3) = 3$$

$$(2 < 1 \rightarrow 3, T \rightarrow \frac{0}{0}) =$$

# Wyrażenia warunkowe

Przykłady:

$$(2 < 1 \rightarrow 4, T \rightarrow 3) = 3$$

$$(2 < 1 \rightarrow \frac{0}{0}, T \rightarrow 3) = 3$$

$$(2 < 1 \rightarrow 3, T \rightarrow \frac{0}{0}) = \textit{undefined}$$

# Wyrażenia warunkowe

Przykłady:

$$(2 < 1 \rightarrow 4, T \rightarrow 3) = 3$$

$$(2 < 1 \rightarrow \frac{0}{0}, T \rightarrow 3) = 3$$

$$(2 < 1 \rightarrow 3, T \rightarrow \frac{0}{0}) = \textit{undefined}$$

$$(2 < 1 \rightarrow 3, 4 < 1 \rightarrow 1) =$$



# Wyrażenia warunkowe

Przykłady:

$$(2 < 1 \rightarrow 4, T \rightarrow 3) = 3$$

$$(2 < 1 \rightarrow \frac{0}{0}, T \rightarrow 3) = 3$$

$$(2 < 1 \rightarrow 3, T \rightarrow \frac{0}{0}) = \textit{undefined}$$

$$(2 < 1 \rightarrow 3, 4 < 1 \rightarrow 1) = \textit{undefined}$$

# Wyrażenia warunkowe

Wyrażenia warunkowe pozwalają na eleganckie zapisywanie funkcji:

# Wyrażenia warunkowe

Wyrażenia warunkowe pozwalają na eleganckie zapisywanie funkcji:

$$|x| = (x < 0 \rightarrow -x, T \rightarrow x)$$

# Wyrażenia warunkowe

Wyrażenia warunkowe pozwalają na eleganckie zapisywanie funkcji:

$$|x| = (x < 0 \rightarrow -x, T \rightarrow x)$$

$$\text{sgn}(x) = (x < 0 \rightarrow -1, x = 0 \rightarrow 0, T \rightarrow 1)$$

# Wyrażenia warunkowe

Wyrażenia warunkowe pozwalają na eleganckie zapisywanie funkcji:

$$|x| = (x < 0 \rightarrow -x, T \rightarrow x)$$

$$\text{sgn}(x) = (x < 0 \rightarrow -1, x = 0 \rightarrow 0, T \rightarrow 1)$$

W „*History of Lisp*” McCarthy narzekał na niewygodne instrukcje warunkowe w FORTRANie.

# Wyrażenia warunkowe

Wyrażenia warunkowe pozwalają na eleganckie zapisywanie funkcji:

$$|x| = (x < 0 \rightarrow -x, T \rightarrow x)$$

$$\text{sgn}(x) = (x < 0 \rightarrow -1, x = 0 \rightarrow 0, T \rightarrow 1)$$

W „*History of Lisp*” McCarthy narzekał na niewygodne instrukcje warunkowe w FORTRANie. Naturalnym było wynalezienie funkcji  $\text{XIF}(M, N1, N2)$ , która zwracała  $N1$  lub  $N2$  zależnie od wartości  $M$ , ale FORTRAN miał ewaluację gorliwą.

# Funkcje rekurencyjne

Zapis funkcji rekurencyjnych przy użyciu wyrażeń warunkowych:

$$n! = (n = 0 \rightarrow 1, T \rightarrow n \cdot (n - 1)!) )$$

# Funkcje rekurencyjne

Zapis funkcji rekurencyjnych przy użyciu wyrażeń warunkowych:

$$n! = (n = 0 \rightarrow 1, T \rightarrow n \cdot (n - 1)!)$$

$$\begin{aligned} \gcd(m, n) &= (m > n \rightarrow \gcd(n, m), \\ &\quad \text{rem}(n, m) = 0 \rightarrow m, \\ &\quad T \rightarrow \gcd(\text{rem}(n, m), m)) \end{aligned}$$



There is no guarantee that the computation determined by a recursive definition will ever terminate and, for example, an attempt to compute  $n!$  from our definition will only succeed if  $n$  is a non-negative integer. If the computation does not terminate, the function must be regarded as undefined for the given arguments.

# Spójniki logiczne

$$p \wedge q = (p \rightarrow q, T \rightarrow F)$$

# Spójniki logiczne

$$p \wedge q = (p \rightarrow q, T \rightarrow F)$$

$$p \vee q = (p \rightarrow T, T \rightarrow q)$$

# Spójniki logiczne

$$p \wedge q = (p \rightarrow q, T \rightarrow F)$$

$$p \vee q = (p \rightarrow T, T \rightarrow q)$$

$$\sim p = (p \rightarrow F, T \rightarrow T)$$

# Spójniki logiczne

$$p \wedge q = (p \rightarrow q, T \rightarrow F)$$

$$p \vee q = (p \rightarrow T, T \rightarrow q)$$

$$\sim p = (p \rightarrow F, T \rightarrow T)$$

$$p \Rightarrow q = (p \rightarrow q, T \rightarrow T)$$

It is readily seen that the right-hand sides of the equations have the correct truth tables. If we consider situations in which  $p$  or  $q$  may be undefined, the connectives  $\wedge$  and  $\vee$  are seen to be noncommutative. For example if  $p$  is false and  $q$  is undefined, we see that according to the definitions given above  $p \wedge q$  is false, but  $q \wedge p$  is undefined. For our applications this noncommutativity is desirable, since  $p \wedge q$  is computed by first computing  $p$ , and if  $p$  is false  $q$  is not computed. If the computation for  $p$  does not terminate, we never get around to computing  $q$ .

Rysunek: Ewaluacja leniwa?

# Spójniki logiczne – leniwa ewaluacja

$$p \wedge q = (p \rightarrow q, T \rightarrow F)$$

Argumenty funkcji „ $\wedge$ ” nie są poddawane ewaluacji przed jej aplikacją.



# Formy a funkcje

# Formy a funkcje

Pojęcie formy zostało zapożyczone z rachunku lambda Churcha.

# Formy a funkcje

Pojęcie formy zostało zapożyczone z rachunku lambda Churcha.

$y^2 + x$  – forma

# Formy a funkcje

Pojęcie formy zostało zapożyczone z rachunku lambda Churcha.

$y^2 + x$  – forma

$(y^2 + x)(3, 4)$  – źle. Nie wiadomo czy wartością powinno być 13 czy 19.

# Formy a funkcje

Pojęcie formy zostało zapożyczone z rachunku lambda Churcha.

$y^2 + x$  – forma

$(y^2 + x)(3, 4)$  – źle. Nie wiadomo czy wartością powinno być 13 czy 19.

Formę  $\mathcal{E}$  możemy skonwertować na funkcję jeżeli ustalimy powiązanie między zmiennymi w formie a uporządkowaną listą argumentów funkcji:

# Formy a funkcje

Pojęcie formy zostało zapożyczone z rachunku lambda Churcha.

$y^2 + x$  – forma

$(y^2 + x)(3, 4)$  – źle. Nie wiadomo czy wartością powinno być 13 czy 19.

Formę  $\mathcal{E}$  możemy skonwertować na funkcję jeżeli ustalimy powiązanie między zmiennymi w formie a uporządkowaną listą argumentów funkcji:

$\lambda((x_1, \dots, x_n), \mathcal{E}))$  – funkcja

# Formy a funkcje

Pojęcie formy zostało zapożyczone z rachunku lambda Churcha.

$y^2 + x$  – forma

$(y^2 + x)(3, 4)$  – źle. Nie wiadomo czy wartością powinno być 13 czy 19.

Formę  $\mathcal{E}$  możemy skonwertować na funkcję jeżeli ustalimy powiązanie między zmiennymi w formie a uporządkowaną listą argumentów funkcji:

$\lambda((x_1, \dots, x_n), \mathcal{E}))$  – funkcja

$\lambda((x, y), y^2 + x)$  – też funkcja

# Rekurencyjne $\lambda$ -wyrażenia



# Rekurencyjne $\lambda$ -wyrażenia

$$\begin{aligned} \text{sqrt} = & \lambda((a, x, \epsilon), \\ & (|x^2 - a| < \epsilon \rightarrow x, T \rightarrow \text{sqrt}(a, \frac{1}{2}(x + \frac{a}{x}), \epsilon))) \end{aligned}$$

# Rekurencyjne $\lambda$ -wyrażenia

$$\begin{aligned} \text{sqrt} = \lambda((a, x, \epsilon), \\ (|x^2 - a| < \epsilon \rightarrow x, T \rightarrow \text{sqrt}(a, \frac{1}{2}(x + \frac{a}{x}), \epsilon))) \end{aligned}$$

Prawa strona wyrażenia nie może być wyrażeniem dla tej funkcji, bo nic nie wskazuje na to, że *sqrt* do odnosi się do całego wyrażenia (*sqrt* nie jest związane).

# Rekurencyjne $\lambda$ -wyrażenia

$$\begin{aligned} \text{sqrt} = \lambda((a, x, \epsilon), \\ (|x^2 - a| < \epsilon \rightarrow x, T \rightarrow \text{sqrt}(a, \frac{1}{2}(x + \frac{a}{x}), \epsilon))) \end{aligned}$$

Prawa strona wyrażenia nie może być wyrażeniem dla tej funkcji, bo nic nie wskazuje na to, że *sqrt* do odnosi się do całego wyrażenia (*sqrt* nie jest związane).

Operator punktu stałego?

# Rekurencyjne $\lambda$ -wyrażenia

$$\begin{aligned} \text{sqrt} = \lambda((a, x, \epsilon), \\ (|x^2 - a| < \epsilon \rightarrow x, T \rightarrow \text{sqrt}(a, \frac{1}{2}(x + \frac{a}{x}), \epsilon))) \end{aligned}$$

Prawa strona wyrażenia nie może być wyrażeniem dla tej funkcji, bo nic nie wskazuje na to, że *sqrt* do odnosi się do całego wyrażenia (*sqrt* nie jest związane).

Operator punktu stałego? – zbyt długie i nieczytelne wyrażenia.

# Rekurencyjne $\lambda$ -wyrażenia

Nowa notacja:

# Rekurencyjne $\lambda$ -wyrażenia

Nowa notacja:

*label*( $a, \mathcal{E}$ )

# Rekurencyjne $\lambda$ -wyrażenia

Nowa notacja:

$$label(a, \mathcal{E})$$

Wyrażenie  $\mathcal{E}$ , w którym wszystkie wystąpienia  $a$  interpretowane są jako odniesienia do całego wyrażenia  $\mathcal{E}$ .

# S-wyrażenia



# S-wyrażenia

„*A Class of Symbolic Expressions*”.

# S-wyrażenia

„*A Class of Symbolic Expressions*”.

Dopuszczalne znaki:

- .

# S-wyrażenia

„*A Class of Symbolic Expressions*”.

Dopuszczalne znaki:

- .
- )

# S-wyrażenia

„*A Class of Symbolic Expressions*”.

Dopuszczalne znaki:

- .
- )
- (

# S-wyrażenia

„*A Class of Symbolic Expressions*”.

Dopuszczalne znaki:

- .
- )
- (
- nieskończony zbiór rozróżnialnych symboli atomowych – napisów złożonych z wielkich liter alfabetu łacińskiego, cyfr i pojedynczych spacji.

# S-wyrażenia

„*A Class of Symbolic Expressions*”.

Dopuszczalne znaki:

- .
- )
- (
- nieskończony zbiór rozróżnialnych symboli atomowych – napisów złożonych z wielkich liter alfabetu łacińskiego, cyfr i pojedynczych spacji.

Na przykład:

- ▶ A
- ▶ ABA
- ▶ APPLE PIE NUMBER 3

# S-wyrażenia

Czemu symbole składające się z wielu znaków?

# S-wyrażenia

Czemu symbole składające się z wielu znaków?

Odpowiedź:

- IBM 704 miał tylko 47 drukowalnych znaków więc jednoliterowe symbole byłyby niewystarczające



Czemu symbole składające się z wielu znaków?

Odpowiedź:

- IBM 704 miał tylko 47 drukowalnych znaków więc jednoliterowe symbole byłyby niewystarczające
- Nazywanie atomowych bytów angielskimi słowami i zdaniami jest wygodne

# S-wyrażenia

Definicja indukcyjna S-wyrażeń:

# S-wyrażenia

Definicja indukcyjna S-wyrażeń:

- 1 Symbole atomowe są S-wyrażeniami

# S-wyrażenia

Definicja indukcyjna S-wyrażeń:

- 1 Symbole atomowe są S-wyrażeniami
- 2 Jeżeli  $e_1$  i  $e_2$  są S-wyrażeniami, to  $(e_1 \cdot e_2)$  również jest S-wyrażeniem.

# S-wyrażenia

Definicja indukcyjna S-wyrażeń:

- 1 Symbole atomowe są S-wyrażeniami
- 2 Jeżeli  $e_1$  i  $e_2$  są S-wyrażeniami, to  $(e_1 \cdot e_2)$  również jest S-wyrażeniem.

Przykładowe S-wyrażenia:

$$AB$$
$$(A \cdot B)$$
$$((AB \cdot C) \cdot D)$$

# Listy (S-wyrażenia)

Możemy reprezentować listy dowolnej długości przy użyciu S-wyrażeń w następujący sposób:

# Listy (S-wyrażenia)

Możemy reprezentować listy dowolnej długości przy użyciu S-wyrażeń w następujący sposób:

$$(m_1, m_2, \dots m_n) = (m_1 \cdot (m_2 \cdot (\dots (m_n \cdot NIL) \dots)))$$

# Listy (S-wyrażenia)

Możemy reprezentować listy dowolnej długości przy użyciu S-wyrażeń w następujący sposób:

$$(m_1, m_2, \dots m_n) = (m_1 \cdot (m_2 \cdot (\dots (m_n \cdot NIL) \dots)))$$

Gdzie NIL jest specjalnym symbolem kończącym listy.



# Listy (S-wyrażenia)

Możemy reprezentować listy dowolnej długości przy użyciu S-wyrażeń w następujący sposób:

$$(m_1, m_2, \dots m_n) = (m_1 \cdot (m_2 \cdot (\dots (m_n \cdot NIL) \dots)))$$

Gdzie NIL jest specjalnym symbolem kończącym listy.

Wprowadzamy zatem specjalną notację dla list:

# Listy (S-wyrażenia)

Możemy reprezentować listy dowolnej długości przy użyciu S-wyrażeń w następujący sposób:

$$(m_1, m_2, \dots m_n) = (m_1 \cdot (m_2 \cdot (\dots (m_n \cdot NIL) \dots)))$$

Gdzie NIL jest specjalnym symbolem kończącym listy.

Wprowadzamy zatem specjalną notację dla list:

- ❶  $(m) = (m \cdot NIL)$
- ❷  $(m_1, m_2, \dots m_n) = (m_1 \cdot (\dots (m_n \cdot NIL) \dots))$
- ❸  $(m_1, \dots, m_n \cdot x) = (m_1 \cdot (\dots (m_n \cdot x) \dots))$

# M-wyrażenia

# M-wyrażenia

Aby odróżnić wyrażenia reprezentujące aplikację funkcji od S-wyrażeń, będziemy małych liter do nazywania funkcji i zmiennych.

# M-wyrażenia

Aby odróżnić wyrażenia reprezentujące aplikację funkcji od S-wyrażeń, będziemy małych liter do nazywania funkcji i zmiennych.

Będziemy również używać nawiasów kwadratowych i średników zamiast nawiasów i przecinków.

# M-wyrażenia

Aby odróżnić wyrażenia reprezentujące aplikację funkcji od S-wyrażeń, będziemy małych liter do nazywania funkcji i zmiennych.

Będziemy również używać nawiasów kwadratowych i średników zamiast nawiasów i przecinków.

Na przykład:

$$car[x]$$
$$car[cons[(A \cdot B); x]]$$

# M-wyrażenia

Aby odróżnić wyrażenia reprezentujące aplikację funkcji od S-wyrażeń, będziemy małych liter do nazywania funkcji i zmiennych.

Będziemy również używać nawiasów kwadratowych i średników zamiast nawiasów i przecinków.

Na przykład:

$$car[x]$$
$$car[cons[(A \cdot B); x]]$$

M-wyrażenia – meta-wyrażenia

# Funkcje i predykaty dla S-wyrażeń



# Funkcje i predykaty dla S-wyrażeń

*atom*[*x*] – prawda tylko jeżeli *x* jest symbolem

*eq*[*x*; *y*] – prawda tylko jeżeli *x* i *y* są tym samym symbolem

*car*[*x*] – zdefiniowany tylko jeżeli *x* nie jest symbolem.

# Funkcje i predykaty dla S-wyrażeń

$atom[x]$  – prawda tylko jeżeli  $x$  jest symbolem

$eq[x; y]$  – prawda tylko jeżeli  $x$  i  $y$  są tym samym symbolem

$car[x]$  – zdefiniowany tylko jeżeli  $x$  nie jest symbolem.

$$car[(X \cdot A)] = X$$

$$car[((X \cdot A) \cdot Y)] = (X \cdot A)$$

# Funkcje i predykaty dla S-wyrażeń

*atom*[*x*] – prawda tylko jeżeli *x* jest symbolem

*eq*[*x*; *y*] – prawda tylko jeżeli *x* i *y* są tym samym symbolem

*car*[*x*] – zdefiniowany tylko jeżeli *x* nie jest symbolem.

$$\text{car}[(X \cdot A)] = X$$

$$\text{car}[((X \cdot A) \cdot Y)] = (X \cdot A)$$

*cdr*[*x*] – zdefiniowany tylko jeżeli *x* nie jest symbolem

# Funkcje i predykaty dla S-wyrażeń

$atom[x]$  – prawda tylko jeżeli  $x$  jest symbolem

$eq[x; y]$  – prawda tylko jeżeli  $x$  i  $y$  są tym samym symbolem

$car[x]$  – zdefiniowany tylko jeżeli  $x$  nie jest symbolem.

$$car[(X \cdot A)] = X$$

$$car[((X \cdot A) \cdot Y)] = (X \cdot A)$$

$cdr[x]$  – zdefiniowany tylko jeżeli  $x$  nie jest symbolem

$$cdr[(X \cdot A)] = A$$

$$cdr[((X \cdot A) \cdot Y)] = Y$$

# Funkcje i predykaty dla S-wyrażeń

$atom[x]$  – prawda tylko jeżeli  $x$  jest symbolem

$eq[x; y]$  – prawda tylko jeżeli  $x$  i  $y$  są tym samym symbolem

$car[x]$  – zdefiniowany tylko jeżeli  $x$  nie jest symbolem.

$$car[(X \cdot A)] = X$$

$$car[((X \cdot A) \cdot Y)] = (X \cdot A)$$

$cdr[x]$  – zdefiniowany tylko jeżeli  $x$  nie jest symbolem

$$cdr[(X \cdot A)] = A$$

$$cdr[((X \cdot A) \cdot Y)] = Y$$

$$cons[x; y] = (e_1 \cdot e_2)$$

# Funkcje i predykaty dla S-wyrażeń

Widać, że *car*, *cdr* i *cons* spełniają poniższe zależności:

# Funkcje i predykaty dla S-wyrażeń

Widać, że *car*, *cdr* i *cons* spełniają poniższe zależności:

$$\text{car}[\text{cons}[x; y]] = x$$

$$\text{cdr}[\text{cons}[x; y]] = y$$

$$\text{cons}[\text{car}[x]; \text{cdr}[x]] = x \text{ (pod warunkiem, że } x \text{ nie jest atomowe)}$$

# Funkcje i predykaty dla S-wyrażeń

Dalej McCarthy opisuje różne funkcje, które można zapisać przy użyciu wcześniej opisanej notacji.

- *append*[*x*; *y*]



# Funkcje i predykaty dla S-wyrażeń

Dalej McCarthy opisuje różne funkcje, które można zapisać przy użyciu wcześniej opisanej notacji.

- *append*[ $x; y$ ]
- *among*[ $x; y$ ]

# Funkcje i predykaty dla S-wyrażeń

Dalej McCarthy opisuje różne funkcje, które można zapisać przy użyciu wcześniej opisanej notacji.

- *append* $[x; y]$
- *among* $[x; y]$  – czy  $y$  zawiera  $x$ ?

# Funkcje i predykaty dla S-wyrażeń

Dalej McCarthy opisuje różne funkcje, które można zapisać przy użyciu wcześniej opisanej notacji.

- *append*[ $x; y$ ]
- *among*[ $x; y$ ] – czy  $y$  zawiera  $x$ ?
- *pair*[ $x; y$ ]

# Funkcje i predykaty dla S-wyrażeń

Dalej McCarthy opisuje różne funkcje, które można zapisać przy użyciu wcześniej opisanej notacji.

- *append*[ $x; y$ ]
- *among*[ $x; y$ ] – czy  $y$  zawiera  $x$ ?
- *pair*[ $x; y$ ] – zip

# Funkcje i predykaty dla S-wyrażeń

Dalej McCarthy opisuje różne funkcje, które można zapisać przy użyciu wcześniej opisanej notacji.

- *append*[ $x; y$ ]
- *among*[ $x; y$ ] – czy  $y$  zawiera  $x$ ?
- *pair*[ $x; y$ ] – zip
- *assoc*[ $x; y$ ]

# Funkcje i predykaty dla S-wyrażeń

Dalej McCarthy opisuje różne funkcje, które można zapisać przy użyciu wcześniej opisanej notacji.

- *append*[ $x; y$ ]
- *among*[ $x; y$ ] – czy  $y$  zawiera  $x$ ?
- *pair*[ $x; y$ ] – zip
- *assoc*[ $x; y$ ] – szukanie wartości dla klucza  $x$  na liście par  $y$

# Funkcje i predykaty dla S-wyrażeń

Dalej McCarthy opisuje różne funkcje, które można zapisać przy użyciu wcześniej opisanej notacji.

- *append* $[x; y]$
- *among* $[x; y]$  – czy  $y$  zawiera  $x$ ?
- *pair* $[x; y]$  – zip
- *assoc* $[x; y]$  – szukanie wartości dla klucza  $x$  na liście par  $y$
- *sublis* $[x; y]$

# Funkcje i predykaty dla S-wyrażeń

Dalej McCarthy opisuje różne funkcje, które można zapisać przy użyciu wcześniej opisanej notacji.

- *append*[ $x; y$ ]
- *among*[ $x; y$ ] – czy  $y$  zawiera  $x$ ?
- *pair*[ $x; y$ ] – zip
- *assoc*[ $x; y$ ] – szukanie wartości dla klucza  $x$  na liście par  $y$
- *sublis*[ $x; y$ ] – mamy listę symboli  $y$ , pod które chcemy podstawić przypisane im wartości z listy asocjacji  $x$



# Funkcje i predykaty dla S-wyrażeń

Dalej McCarthy opisuje różne funkcje, które można zapisać przy użyciu wcześniej opisanej notacji.

- *append*[ $x; y$ ]
- *among*[ $x; y$ ] – czy  $y$  zawiera  $x$ ?
- *pair*[ $x; y$ ] – zip
- *assoc*[ $x; y$ ] – szukanie wartości dla klucza  $x$  na liście par  $y$
- *sublis*[ $x; y$ ] – mamy listę symboli  $y$ , pod które chcemy podstawić przypisane im wartości z listy asocjacji  $x$

# Tłumaczenie M-wyrażeń na S-wyrażenia

Chcemy przetłumaczyć M-wyrażenie  $\mathcal{E}$  na S-wyrażenie  $\mathcal{E}^*$

# Tłumaczenie M-wyrażeń na S-wyrażenia

Chcemy przetłumaczyć M-wyrażenie  $\mathcal{E}$  na S-wyrażenie  $\mathcal{E}^*$

- 1 Jeżeli  $\mathcal{E}$  jest S-wyrażeniem, to  $\mathcal{E}^* = (QUOTE, \mathcal{E})$

# Tłumaczenie M-wyrażeń na S-wyrażenia

Chcemy przetłumaczyć M-wyrażenie  $\mathcal{E}$  na S-wyrażenie  $\mathcal{E}^*$

- 1 Jeżeli  $\mathcal{E}$  jest S-wyrażeniem, to  $\mathcal{E}^* = (QUOTE, \mathcal{E})$
- 2 Zmienne i nazwy funkcji napisane małymi literami zamieniamy na odpowiadające napisy złożone z wielkich liter (np.  $car^* = CAR$ ).

# Tłumaczenie M-wyrażeń na S-wyrażenia

Chcemy przetłumaczyć M-wyrażenie  $\mathcal{E}$  na S-wyrażenie  $\mathcal{E}^*$

- 1 Jeżeli  $\mathcal{E}$  jest S-wyrażeniem, to  $\mathcal{E}^* = (QUOTE, \mathcal{E})$
- 2 Zmienne i nazwy funkcji napisane małymi literami zamieniamy na odpowiadające napisy złożone z wielkich liter (np.  $car^* = CAR$ ).
- 3  $\{f[e_1; \dots; e_n]\}^* = (f^*, e_1^*, \dots, e_n^*)$

# Tłumaczenie M-wyrażeń na S-wyrażenia

Chcemy przetłumaczyć M-wyrażenie  $\mathcal{E}$  na S-wyrażenie  $\mathcal{E}^*$

- 1 Jeżeli  $\mathcal{E}$  jest S-wyrażeniem, to  $\mathcal{E}^* = (QUOTE, \mathcal{E})$
- 2 Zmienne i nazwy funkcji napisane małymi literami zamieniamy na odpowiadające napisy złożone z wielkich liter (np.  $car^* = CAR$ ).
- 3  $\{f[e_1; \dots; e_n]\}^* = (f^*, e_1^*, \dots, e_n^*)$
- 4  $\{[p_1 \rightarrow e_1; \dots; p_n \rightarrow e_n]\}^* = (COND, (p_1^*, e_1^*), \dots (p_n^*, e_n^*))$

# Tłumaczenie M-wyrażeń na S-wyrażenia

Chcemy przetłumaczyć M-wyrażenie  $\mathcal{E}$  na S-wyrażenie  $\mathcal{E}^*$

- 1 Jeżeli  $\mathcal{E}$  jest S-wyrażeniem, to  $\mathcal{E}^* = (QUOTE, \mathcal{E})$
- 2 Zmienne i nazwy funkcji napisane małymi literami zamieniamy na odpowiadające napisy złożone z wielkich liter (np.  $car^* = CAR$ ).
- 3  $\{f[e_1; \dots; e_n]\}^* = (f^*, e_1^*, \dots, e_n^*)$
- 4  $\{[p_1 \rightarrow e_1; \dots; p_n \rightarrow e_n]\}^* = (COND, (p_1^*, e_1^*), \dots (p_n^*, e_n^*))$
- 5  $\{[\lambda[[x_1; \dots; x_n]; \mathcal{E}]]\}^* = (LAMBDA, (x_1^*, \dots, x_n^*), \mathcal{E}^*)$

# Tłumaczenie M-wyrażeń na S-wyrażenia

Chcemy przetłumaczyć M-wyrażenie  $\mathcal{E}$  na S-wyrażenie  $\mathcal{E}^*$

- 1 Jeżeli  $\mathcal{E}$  jest S-wyrażeniem, to  $\mathcal{E}^* = (QUOTE, \mathcal{E})$
- 2 Zmienne i nazwy funkcji napisane małymi literami zamieniamy na odpowiadające napisy złożone z wielkich liter (np.  $car^* = CAR$ ).
- 3  $\{f[e_1; \dots; e_n]\}^* = (f^*, e_1^*, \dots, e_n^*)$
- 4  $\{[p_1 \rightarrow e_1; \dots; p_n \rightarrow e_n]\}^* = (COND, (p_1^*, e_1^*), \dots (p_n^*, e_n^*))$
- 5  $\{[\lambda[[x_1; \dots; x_n]; \mathcal{E}]]\}^* = (LAMBDA, (x_1^*, \dots, x_n^*), \mathcal{E}^*)$
- 6  $label[a; \mathcal{E}]^* = (LABEL, a^*, \mathcal{E}^*)$



```
(LABEL, SUEST, (LAMBDA, (X, Y, Z), (COND  
  ((ATOM, Z), (COND, (EQ, Y, Z), X), ((QUOTE,  
    T), Z))), ((QUOTE, T), (CONS, (SUBST, X, Y,  
    (CAR Z)), (SUBST, X, Y, (CDR, Z))))))
```

```
(LABEL, SUEST, (LAMBDA, (X, Y, Z), (COND
  ((ATOM, Z), (COND, (EQ, Y, Z), X), ((QUOTE,
    T), Z))), ((QUOTE, T), (CONS, (SUBST, X, Y,
      (CAR Z)), (SUBST, X, Y, (CDR, Z))))))
```

This notation is writable and somewhat readable. It can be made easier to read and write at the cost of making its structure less regular. If more characters were available on the computer, it could be improved considerably.

# *apply*

Funkcja *apply* aplikuje S-wyrażenie  $f$  reprezentujące S-funkcję  $f'$  do listy argumentów *args* postaci  $(arg_1, \dots arg_n)$ .

## *apply*

Funkcja *apply* aplikuje S-wyrażenie  $f$  reprezentujące S-funkcję  $f'$  do listy argumentów  $args$  postaci  $(arg_1, \dots arg_n)$ .

$$apply[f; args] = eval[cons[f; appq[args]]; NIL]$$

gdzie

$$\begin{aligned} appq[m] = \\ [null[m] \rightarrow NIL; \\ T \rightarrow cons[list[QUOTE; car[m]]; appq[cdr[m]]]] \end{aligned}$$

*eval*

$eval[e; a] = [$

$$\text{eval}[e; a] = [$$
$$\text{atom}[e] \rightarrow \text{assoc}[e; a]$$

# *eval*

```
eval[e; a] = [  
  atom[e] → assoc[e; a]  
  atom[car[e]] → [  
    eval[car[cdr[e]]; a]  
  ]  
]
```

```
eval[e; a] = [  
  atom[e] → assoc[e; a]  
  atom[car[e]] → [  
    eq[car[e]; QUOTE] → cadr[e];
```



```
eval[e; a] = [  
  atom[e] → assoc[e; a]  
  atom[car[e]] → [  
    eq[car[e]; QUOTE] → cadr[e];  
    eq[car[e]; ATOM] → atom[eval[cadr[e]; a]];
```

```
eval[e; a] = [  
  atom[e] → assoc[e; a]  
  atom[car[e]] → [  
    eq[car[e]; QUOTE] → cadr[e];  
    eq[car[e]; ATOM] → atom[eval[cadr[e]; a]];  
    eq[car[e]; EQ] → eval[cadr[e]; a] = eval[caddr[e]; a];
```

```
eval[e; a] = [  
  atom[e] → assoc[e; a]  
  atom[car[e]] → [  
    eq[car[e]; QUOTE] → cadr[e];  
    eq[car[e]; ATOM] → atom[eval[cadr[e]; a]];  
    eq[car[e]; EQ] → eval[cadr[e]; a] = eval[caddr[e]; a];  
    eq[car[e]; COND] → evcon[cdr[e]; a];
```

```
eval[e; a] = [  
  atom[e] → assoc[e; a]  
  atom[car[e]] → [  
    eq[car[e]; QUOTE] → cadr[e];  
    eq[car[e]; ATOM] → atom[eval[cadr[e]; a]];  
    eq[car[e]; EQ] → eval[cadr[e]; a] = eval[caddr[e]; a];  
    eq[car[e]; COND] → evcon[cdr[e]; a];  
    eq[car[e]; CAR] → car[eval[cadr[e]; a]];
```

```
eval[e; a] = [  
  atom[e] → assoc[e; a]  
  atom[car[e]] → [  
    eq[car[e]; QUOTE] → cadr[e];  
    eq[car[e]; ATOM] → atom[eval[cadr[e]; a]];  
    eq[car[e]; EQ] → eval[cadr[e]; a] = eval[caddr[e]; a];  
    eq[car[e]; COND] → evcon[cdr[e]; a];  
    eq[car[e]; CAR] → car[eval[cadr[e]; a]];  
    eq[car[e]; CDR] → cdr[eval[cadr[e]; a]];
```

```
eval[e; a] = [  
  atom[e] → assoc[e; a]  
  atom[car[e]] → [  
    eq[car[e]; QUOTE] → cadr[e];  
    eq[car[e]; ATOM] → atom[eval[cadr[e]; a]];  
    eq[car[e]; EQ] → eval[cadr[e]; a] = eval[caddr[e]; a];  
    eq[car[e]; COND] → evcon[cdr[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]];
```

```
eval[e; a] = [  
  atom[e] → assoc[e; a]  
  atom[car[e]] → [  
    eq[car[e]; QUOTE] → cadr[e];  
    eq[car[e]; ATOM] → atom[eval[cadr[e]; a]];  
    eq[car[e]; EQ] → eval[cadr[e]; a] = eval[caddr[e]; a];  
    eq[car[e]; COND] → evcon[cdr[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]];  
    T → eval[cons[assoc[car[e]; a]; evlis[cdr[e]; a]]; a]];
```

*eval* cd.

```
eq[caar[e]; LABEL] →  
  eval[cons[caddar[e]; cdr[e]];  
  cons[list[cadar[e]; car[e]; a]];
```



*eval* cd.

```
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]]  
]
```

*eval* *cd*.

```
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]]  
]
```

gdzie

$$\text{evcon}[e; a] = [\text{eval}[\text{caar}[e]; a] \rightarrow \text{eval}[\text{cadar}[e]; a]; T \rightarrow \text{evcon}[\text{cdr}[e]; a]]$$
$$\text{evlis}[m; a] = [\text{null}[m] \rightarrow \text{NIL}; T \rightarrow \text{cons}[\text{eval}[\text{car}[m]; a]; \text{evlis}[\text{cdr}[m]; a]]]$$

# Funkcje wyższego rzędu

# Funkcje wyższego rzędu

g. *Functions with Functions as Arguments.* There are a number of useful functions some of whose arguments are functions. They are especially useful in defining other functions.

# Funkcje wyższego rzędu

g. *Functions with Functions as Arguments.* There are a number of useful functions some of whose arguments are functions. They are especially useful in defining other functions.

$$\begin{aligned} \text{maplist}[x; f] = [ \\ \text{null}[x] \rightarrow \text{NIL}; \\ T \rightarrow \text{cons}[f[x]; \text{maplist}[\text{cdr}[x]; f]]] \end{aligned}$$

## Funkcje wyższego rzędu

g. *Functions with Functions as Arguments.* There are a number of useful functions some of whose arguments are functions. They are especially useful in defining other functions.

$$\begin{aligned} \text{maplist}[x; f] = [ \\ \text{null}[x] \rightarrow \text{NIL}; \\ T \rightarrow \text{cons}[f[x]; \text{maplist}[\text{cdr}[x]; f]]] \end{aligned}$$
$$\begin{aligned} \text{search}[x; p; f; u] = [ \\ \text{null}[x] \rightarrow u; \\ p[x] \rightarrow f[x]; \\ T \rightarrow \text{search}[\text{cdr}[x]; p; f; u]] \end{aligned}$$

```
diff [y; x] = [atom [y] → [eq [y; x] → ONE; T → ZERO];
eq [car [y]; PLUS] → cons [PLUS; maplist [cdr [y]; λ[z];
diff[car [z]; x]]]; eq[car [y]; TIMES] → cons[PLUS;
maplist[cdr[y]; λ[z]; cons [TIMES; maplist{cdr [y];
λ[[w]; ~eq [z; w] → car [w]; T → diff [car [[w]; x]]]]]]]
```

The derivative of the allowed expression, as computed by this formula, is

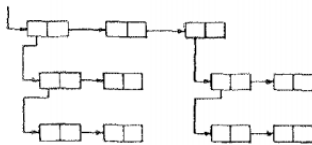
```
(PLUS, (TIMES, ONE, (PLUS, X, A), Y),
      (TIMES, X, (PLUS, ONE, ZERO), Y),
      (TIMES, X, (PLUS, X, A), ZERO))
```

**Rysunek:** Funkcja obliczająca pochodne funkcji zaaplikowana do wyrażenia  
 $(TIMES, X, (PLUS, X, A), Y)$

# Reprezentacja S-wyrażeń w pamięci

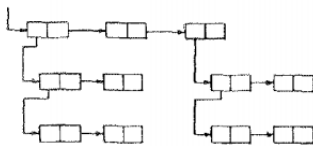


# Reprezentacja S-wyrażeń w pamięci



Lista jest zbiorem słów maszynowych połączonych wskaźnikami w jedną strukturę. Każde słowo na rysunku jest reprezentowane jako jeden z podzielonych na pół prostokątów.

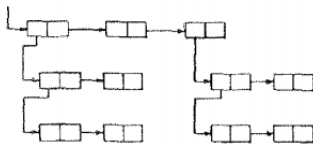
# Reprezentacja S-wyrażeń w pamięci



Lista jest zbiorem słów maszynowych połączonych wskaźnikami w jedną strukturę. Każde słowo na rysunku jest reprezentowane jako jeden z podzielonych na pół prostokątów.

Słowo maszynowe w IBM 704 dzieli się na dwa pola: lewe *address* oraz prawe *decrement* i 6 pozostałych bitów.

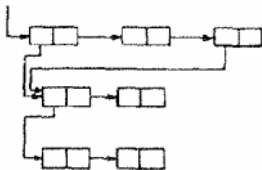
# Reprezentacja S-wyrażeń w pamięci



Lista jest zbiorem słów maszynowych połączonych wskaźnikami w jedną strukturę. Każde słowo na rysunku jest reprezentowane jako jeden z podzielonych na pół prostokątów.

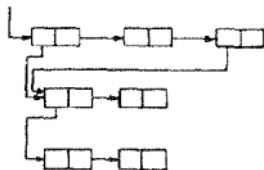
Słowo maszynowe w IBM 704 dzieli się na dwa pola: lewe *address* oraz prawe *decrement* i 6 pozostałych bitów.

# Reprezentacja S-wyrażeń w pamięci

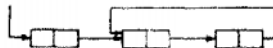


Jedna podstruktura może pojawiać się więcej niż w jednym miejscu w strukturze...

# Reprezentacja S-wyrażeń w pamięci



Jedna podstruktura może pojawiać się więcej niż w jednym miejscu w strukturze...



...ale struktury nie mogą zawierać cykliów.

The prohibition against circular list structures is essentially a prohibition against an expression being a sub-expression of itself. Such an expression could not exist on paper in a world with our topology. Circular list structures would have some advantages in the machine, for example, for representing recursive functions, but difficulties in printing them, and in certain other operations, make it seem advisable not to use them for the present.

# Reprezentacja S-wyrażeń cd.

Benefity wynikające z używania list:

# Reprezentacja S-wyrażeń cd.

Benefity wynikające z używania list:

- Ani rozmiar, ani liczba wyrażeń w programie nie może być przewidziana wcześniej, więc ciężko byłoby zaaranżować bloki pamięci o stałym rozmiarze.



# Reprezentacja S-wyrażeń cd.

Benefity wynikające z używania list:

- Ani rozmiar, ani liczba wyrażeń w programie nie może być przewidziana wcześniej, więc ciężko byłoby zaaranżować bloki pamięci o stałym rozmiarze.
- Rejestry można odkładać na listę dostępnej pamięci gdy nie są już potrzebne. Nawet jeden wolny rejestr jest wartościowy w przeciwieństwie do sytuacji gdzie wyrażenia są przechowywane liniowo w ciągłych obszarach pamięci.

# Reprezentacja S-wyrażeń cd.

Benefity wynikające z używania list:

- Ani rozmiar, ani liczba wyrażeń w programie nie może być przewidziana wcześniej, więc ciężko byłoby zaaranżować bloki pamięci o stałym rozmiarze.
- Rejestry można odkładać na listę dostępnej pamięci gdy nie są już potrzebne. Nawet jeden wolny rejestr jest wartościowy w przeciwieństwie do sytuacji gdzie wyrażenia są przechowywane liniowo w ciągłych obszarach pamięci.
- Wyrażenie, które jest podwyrażeniem wielu wyrażeń, może być dzielone i przechowywane jedynie raz.

# Listy asocjacji

# Listy asocjacji

- Każdy symbol może posiadać listę dowolnych dodatkowych informacji z nim powiązanych.

# Listy asocjacji

- Każdy symbol może posiadać listę dowolnych dodatkowych informacji z nim powiązanych.
  - ▶ *print name* – napis reprezentujący symbol na zewnątrz maszyny

# Listy asocjacji

- Każdy symbol może posiadać listę dowolnych dodatkowych informacji z nim powiązanych.
  - ▶ *print name* – napis reprezentujący symbol na zewnątrz maszyny
  - ▶ numeryczna wartość dla symboli, które są liczbami

# Listy asocjacji

- Każdy symbol może posiadać listę dowolnych dodatkowych informacji z nim powiązanych.
  - ▶ *print name* – napis reprezentujący symbol na zewnątrz maszyny
  - ▶ numeryczna wartość dla symboli, które są liczbami
  - ▶ inne S-wyrażenie

# Listy asocjacji

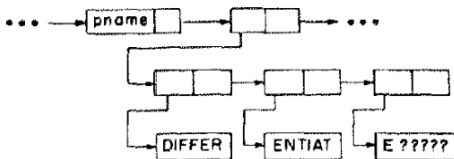
- Każdy symbol może posiadać listę dowolnych dodatkowych informacji z nim powiązanych.
  - ▶ *print name* – napis reprezentujący symbol na zewnątrz maszyny
  - ▶ numeryczna wartość dla symboli, które są liczbami
  - ▶ inne S-wyrażenie
  - ▶ adres procedury dla symboli, które są nazwami dla funkcji zaimplementowanych w kodzie maszynowym



# Listy asocjacji

- Każdy symbol może posiadać listę dowolnych dodatkowych informacji z nim powiązanych.
  - ▶ *print name* – napis reprezentujący symbol na zewnątrz maszyny
  - ▶ numeryczna wartość dla symboli, które są liczbami
  - ▶ inne S-wyrażenie
  - ▶ adres procedury dla symboli, które są nazwami dla funkcji zaimplementowanych w kodzie maszynowym

*Print names* są reprezentowane jako listy słów zawierających maksymalnie sześć 6-bitowych znaków. Ostatnie słowo dopełnione jest wartościami, które nie reprezentują drukowalnych znaków.



Rysunek: *pname* dla symbolu DIFFERENTIATE

# IBM 704

# IBM 704

Kilka faktów o IBM 704:

# IBM 704

Kilka faktów o IBM 704:

- 36-bitowe słowa

# IBM 704

Kilka faktów o IBM 704:

- 36-bitowe słowa
- 15-bitowa przestrzeń adresowa

# IBM 704

Kilka faktów o IBM 704:

- 36-bitowe słowa
- 15-bitowa przestrzeń adresowa
- 38-bitowy akumulator

# IBM 704

Kilka faktów o IBM 704:

- 36-bitowe słowa
- 15-bitowa przestrzeń adresowa
- 38-bitowy akumulator
- 36-bitowy rejestr *multiplier quotient*

# IBM 704

Kilka faktów o IBM 704:

- 36-bitowe słowa
- 15-bitowa przestrzeń adresowa
- 38-bitowy akumulator
- 36-bitowy rejestr *multiplier quotient*
- Trzy 15-bitowe rejestry *index*



# IBM 704

Kilka faktów o IBM 704:

- 36-bitowe słowa
- 15-bitowa przestrzeń adresowa
- 38-bitowy akumulator
- 36-bitowy rejestr *multiplier quotient*
- Trzy 15-bitowe rejestry *index*
- Instrukcje miały dwa typy:

# IBM 704

Kilka faktów o IBM 704:

- 36-bitowe słowa
- 15-bitowa przestrzeń adresowa
- 38-bitowy akumulator
- 36-bitowy rejestr *multiplier quotient*
- Trzy 15-bitowe rejestry *index*
- Instrukcje miały dwa typy:

B – nieciekawe

# IBM 704

Kilka faktów o IBM 704:

- 36-bitowe słowa
- 15-bitowa przestrzeń adresowa
- 38-bitowy akumulator
- 36-bitowy rejestr *multiplier quotient*
- Trzy 15-bitowe rejestry *index*
- Instrukcje miały dwa typy:
  - B – nieciekawe
  - A – każda instrukcja typu A składa się z:

# IBM 704

Kilka faktów o IBM 704:

- 36-bitowe słowa
- 15-bitowa przestrzeń adresowa
- 38-bitowy akumulator
- 36-bitowy rejestr *multiplier quotient*
- Trzy 15-bitowe rejestry *index*
- Instrukcje miały dwa typy:
  - B – nieciekawe
  - A – każda instrukcja typu A składa się z:
    - ▶ 3-bitowy prefiks (kod instrukcji)

## Kilka faktów o IBM 704:

- 36-bitowe słowa
- 15-bitowa przestrzeń adresowa
- 38-bitowy akumulator
- 36-bitowy rejestr *multiplier quotient*
- Trzy 15-bitowe rejestry *index*
- Instrukcje miały dwa typy:
  - B – nieciekawe
  - A – każda instrukcja typu A składa się z:
    - ▶ 3-bitowy prefiks (kod instrukcji)
    - ▶ 15-bitowe pole *decrement* (niektóre instrukcje odejmują tę wartość od wartości w rejestrze)

## Kilka faktów o IBM 704:

- 36-bitowe słowa
- 15-bitowa przestrzeń adresowa
- 38-bitowy akumulator
- 36-bitowy rejestr *multiplier quotient*
- Trzy 15-bitowe rejestry *index*
- Instrukcje miały dwa typy:
  - B – nieciekawe
  - A – każda instrukcja typu A składa się z:
    - ▶ 3-bitowy prefiks (kod instrukcji)
    - ▶ 15-bitowe pole *decrement* (niektóre instrukcje odejmują tę wartość od wartości w rejestrze)
    - ▶ 3-bitowe pole *tag* (wskazuje rejestr)

# IBM 704

Kilka faktów o IBM 704:

- 36-bitowe słowa
- 15-bitowa przestrzeń adresowa
- 38-bitowy akumulator
- 36-bitowy rejestr *multiplier quotient*
- Trzy 15-bitowe rejestry *index*
- Instrukcje miały dwa typy:

B – nieciekawe

A – każda instrukcja typu A składa się z:

- ▶ 3-bitowy prefiks (kod instrukcji)
- ▶ 15-bitowe pole *decrement* (niektóre instrukcje odejmują tę wartość od wartości w rejestrze)
- ▶ 3-bitowe pole *tag* (wskazuje rejestr)
- ▶ 15-bitowe pole *address*

# Free-Storage List



## Free-Storage List

Specjalny rejestr FREE zawiera adres pierwszego słowa listy *free storage*.

## Free-Storage List

Specjalny rejestr FREE zawiera adres pierwszego słowa listy *free storage*.

Jak przebiega alokacja pamięci?

## Free-Storage List

Specjalny rejestr FREE zawiera adres pierwszego słowa listy *free storage*.

Jak przebiega alokacja pamięci?

- 1 Weź pierwsze wolne słowo z listy

## Free-Storage List

Specjalny rejestr FREE zawiera adres pierwszego słowa listy *free storage*.

Jak przebiega alokacja pamięci?

- 1 Weź pierwsze wolne słowo z listy
- 2 Ustaw wartość rejestru FREE na adres drugiego słowa z listy

## *Free-Storage List*

Istnieje ustalony zbiór bazowych rejestrów, które zawierają adresy struktur listowych, do których istnieje dostęp w programie.

## Free-Storage List

Istnieje ustalony zbiór bazowych rejestrów, które zawierają adresy struktur listowych, do których istnieje dostęp w programie.

Każdy rejestr, który jest osiągalny przez program, jest osiągalny z jednego lub więcej bazowych rejestrów poprzez łańcuch operacji *car* i *cdr*.

## Free-Storage List

Istnieje ustalony zbiór bazowych rejestrów, które zawierają adresy struktur listowych, do których istnieje dostęp w programie.

Każdy rejestr, który jest osiągalny przez program, jest osiągalny z jednego lub więcej bazowych rejestrów poprzez łańcuch operacji *car* i *cdr*.

Zmiana zawartości jednego z bazowych rejestrów może spowodować, że wskazywany dotychczas rejestr staje się nieosiągalny. Oznacza to wówczas, że jego zawartość nie jest już potrzebna.

## Free-Storage List

Istnieje ustalony zbiór bazowych rejestrów, które zawierają adresy struktur listowych, do których istnieje dostęp w programie.

Każdy rejestr, który jest osiągalny przez program, jest osiągalny z jednego lub więcej bazowych rejestrów poprzez łańcuch operacji *car* i *cdr*.

Zmiana zawartości jednego z bazowych rejestrów może spowodować, że wskazywany dotychczas rejestr staje się nieosiągalny. Oznacza to wówczas, że jego zawartość nie jest już potrzebna.

Nie jest podejmowane żadne działanie w celu zwalniania pamięci, dopóki programowi nie skończy się cała wolna pamięć. Gdy potrzebny jest wolny rejestr, rozpoczyna się *reclamation cycle*.



# Garbage collector

## Garbage collector

- 1 Znajdujemy wszystkie słowa osiągalne z rejestrów bazowych i ustawiamy ich znak na ujemny.

## Garbage collector (mark-and-sweep)

- 1 Znajdujemy wszystkie słowa osiągalne z rejestrów bazowych i ustawiamy ich znak na ujemny.
- 2 Przeszukujemy obszar pamięci zarezerwowany dla struktur listowych i dodajemy wszystkie słowa o znaku dodatnim do listy *free-storage*.

## Garbage collector (mark-and-sweep)

- 1 Znajdujemy wszystkie słowa osiągalne z rejestrów bazowych i ustawiamy ich znak na ujemny.
- 2 Przeszukujemy obszar pamięci zarezerwowany dla struktur listowych i dodajemy wszystkie słowa o znaku dodatnim do listy *free-storage*.

Implementacja odświeżania pamięci przez zliczanie referencji nie była praktyczna, bo dostępne było jedynie 6 bitów na każde słowo.

# Implementacja wbudowanych funkcji

# Implementacja wbudowanych funkcji

`atom` – słowo reprezentujące symbol ma specjalną stałą w części adresowej.

# Implementacja wbudowanych funkcji

- `atom` – słowo reprezentujące symbol ma specjalną stałą w części adresowej.
- `eq` – numeryczna równość adresów słów. Dla symboli działa to dlatego, że każdy symbol ma tylko jedną listę asocjacji.

# Implementacja wbudowanych funkcji

- atom** – słowo reprezentujące symbol ma specjalną stałą w części adresowej.
- eq** – numeryczna równość adresów słów. Dla symboli działa to dlatego, że każdy symbol ma tylko jedną listę asocjacji.
- cons** – wartością *cons*[*x*; *y*] musi być adres rejestru, który ma adres *x* jako część *address* i *y* jako część *decrement*.  
*Cons* zawsze bierze nowe słowo z listy *free-storage*.  
Wywołanie *cons* może uruchomić garbage collector.



## car i cdr

IBM 704 miał specjalne instrukcje do odczytywania pól *address* i *decrement* słowa. Pozwalało to na wydajne przechowywanie dwóch 15-bitowych wskaźników w jednym słowie.

## car i cdr

IBM 704 miał specjalne instrukcje do odczytywania pól *address* i *decrement* słowa. Pozwalało to na wydajne przechowywanie dwóch 15-bitowych wskaźników w jednym słowie.

Makra car i cdr w assemblerze 704:

# car:

```
CLA 0, i # wyczyść AC i dodaj i
PAX 0, i # zapisz Address AC do i
PXD 0, i # wyczyść AC i wczytaj
        # i do Decrement AC
```

# cdr:

```
CLA 0, i
PDX 0, i
PXD 0, i
```

**car** – „Contents of the **A**ddress **R**egister”

**cdr** – „Contents of the **D**ecrement **R**egister”

*„Because of an unfortunate temporary lapse of inspiration, we couldn't think of any other names for the 2 pointers in a list node than 'address' and 'decrement', so we called the functions CAR for 'Contents of Address of Register' and CDR for 'Contents of Decrement of Register'. After several months and giving a few classes in LISP, we realized that 'first' and 'rest' were better names, and we (John McCarthy, I and some of the rest of the AI Project) tried to get people to use them instead. Alas, it was too late! We couldn't make it stick at all. So we have CAR and CDR.”*

— Steve Russell

# Steve Russell

## Steve Russell

*„S.R. Russell noticed that eval could serve as an interpreter for LISP, promptly hand coded it, and we now had a programming language with an interpreter.”*

— John McCarthy

# Steve Russell

*„S.R. Russell noticed that eval could serve as an interpreter for LISP, promptly hand coded it, and we now had a programming language with an interpreter.”*

— John McCarthy

- Napisał pierwszy interpreter LISPu



# Steve Russell

*„S.R. Russell noticed that eval could serve as an interpreter for LISP, promptly hand coded it, and we now had a programming language with an interpreter.”*

— John McCarthy



- Napisał pierwszy interpreter LISPu
- W 1962 zaprojektował i napisał grę „Spacewar!”



# Wiązanie dynamiczne zmiennych

To, że LISP używa wiązania dynamicznego zmiennych wyszło na jaw (przypadkiem), gdy James R. Slagle napisał poniższą funkcję, i okazało się, że nie działa:



# Wiązanie dynamiczne zmiennych

To, że LISP używa wiązania dynamicznego zmiennych wyszło na jaw (przypadkiem), gdy James R. Slagle napisał poniższą funkcję, i okazało się, że nie działa:

```
testr[x, p, f, u]  $\leftarrow$  if p[x] then f[x]  
    else if atom[x] then u[]  
    else testr[cdr[x], p, f,  $\lambda$  : testr[car[x], p, f, u]]
```

# Wiązanie dynamiczne zmiennych

To, że LISP używa wiązania dynamicznego zmiennych wyszło na jaw (przypadkiem), gdy James R. Slagle napisał poniższą funkcję, i okazało się, że nie działa:

$$\begin{aligned} \text{testr}[x, p, f, u] \leftarrow & \text{if } p[x] \text{ then } f[x] \\ & \text{else if } \text{atom}[x] \text{ then } u[] \\ & \text{else } \text{testr}[\text{cdr}[x], p, f, \lambda : \text{testr}[\underline{\text{car}}[x], p, f, u]] \end{aligned}$$

Wbrew oczekiwaniom, wartość  $\text{car}[x]$  z budowanej kontynuacji  $u$  nie pochodziła z zewnętrznego wywołania rekurencyjnego, ale z wewnętrznego, gdzie funkcja  $u$  została wywołana.

# Wiązanie dynamiczne zmiennych

McCarthy powyższe zachowanie uznał za błąd i liczył na to, że Steve Russell go naprawi.

# Wiązanie dynamiczne zmiennych

McCarthy powyższe zachowanie uznał za błąd i liczył na to, że Steve Russell go naprawi.

Rozwiązanie:

# Wiązanie dynamiczne zmiennych

McCarthy powyższe zachowanie uznał za błąd i liczył na to, że Steve Russell go naprawi.

Rozwiązanie: „FUNARG device” – przekazywanie lokalnego środowiska razem z funkcją.

# Wiązanie dynamiczne zmiennych

McCarthy powyższe zachowanie uznał za błąd i liczył na to, że Steve Russell go naprawi.

Rozwiązanie: „FUNARG device” – przekazywanie lokalnego środowiska razem z funkcją.

Wniosek?

# Wiązanie dynamiczne zmiennych

McCarthy powyższe zachowanie uznał za błąd i liczył na to, że Steve Russell go naprawi.

Rozwiązanie: „FUNARG device” – przekazywanie lokalnego środowiska razem z funkcją.

Wniosek? Wiazanie dynamiczne to bug a nie feature.

## Co po LISPie 1 i 1.5?

**LISP 2** – język łączący semantykę LISPu i składnię Algola. Tworzony wspólnie przez Systems Development Corporation (pierwsza firma wytwarzająca oprogramowanie) i Information International Inc. Planowano implementację na komputer Q32 budowany przez IBM dla celów wojskowych (48-bitowe słowa i 18-bitowe adresy).



# Co po LISPie 1 i 1.5?

**LISP 2** – język łączący semantykę LISPu i składnię Algola. Tworzony wspólnie przez Systems Development Corporation (pierwsza firma wytwarzająca oprogramowanie) i Information International Inc. Planowano implementację na komputer Q32 budowany przez IBM dla celów wojskowych (48-bitowe słowa i 18-bitowe adresy). Gdy okazało się, że Q32 wyposażony będzie w nie więcej niż 48K słów pamięci, zdecydowano się zaimplementować język na komputery IBM 360/67 i PDP-6.

## Co po LISPie 1 i 1.5?

**LISP 2** – język łączący semantykę LISPu i składnię Algola. Tworzony wspólnie przez Systems Development Corporation (pierwsza firma wytwarzająca oprogramowanie) i Information International Inc. Planowano implementację na komputer Q32 budowany przez IBM dla celów wojskowych (48-bitowe słowa i 18-bitowe adresy). Gdy okazało się, że Q32 wyposażony będzie w nie więcej niż 48K słów pamięci, zdecydowano się zaimplementować język na komputery IBM 360/67 i PDP-6. W związku z m.in. wysokimi kosztami projekt został porzucony.

# Co po LISPie 1 i 1.5?

MacLisp – rozwijany na MIT. Nazwa pochodzi od Projektu MAC.

# Co po LISPie 1 i 1.5?

**MacLisp** – rozwijany na MIT. Nazwa pochodzi od Projektu MAC.

- Wprowadzono „*value cell*” w celu przyspieszenia dostępu do zmiennych (w oryginalnym LISPie przeszukiwano liniowo listę asocjacji)

# Co po LISPie 1 i 1.5?

MacLisp – rozwijany na MIT. Nazwa pochodzi od Projektu MAC.

- Wprowadzono „*value cell*” w celu przyspieszenia dostępu do zmiennych (w oryginalnym LISPie przeszukiwano liniowo listę asocjacji)
- DEFMACRO

# Co po LISPie 1 i 1.5?

MacLisp – rozwijany na MIT. Nazwa pochodzi od Projektu MAC.

- Wprowadzono „*value cell*” w celu przyspieszenia dostępu do zmiennych (w oryginalnym LISPie przeszukiwano liniowo listę asocjacji)
- DEFMACRO
- LEXPR – funkcje przyjmujące dowolną liczbę argumentów przekazywanych przez stos

# Co po LISPie 1 i 1.5?

**MacLisp** – rozwijany na MIT. Nazwa pochodzi od Projektu MAC.

- Wprowadzono „*value cell*” w celu przyspieszenia dostępu do zmiennych (w oryginalnym LISPie przeszukiwano liniowo listę asocjacji)
- DEFMACRO
- LEXPR – funkcje przyjmujące dowolną liczbę argumentów przekazywanych przez stos
- ERRSET – funkcja pełniąca rolę try (w 1972 wprowadzono TRY i CATCH)

# Co po LISPie 1 i 1.5?

**MacLisp** – rozwijany na MIT. Nazwa pochodzi od Projektu MAC.

- Wprowadzono „*value cell*” w celu przyspieszenia dostępu do zmiennych (w oryginalnym LISPie przeszukiwano liniowo listę asocjacji)
- DEFMACRO
- LEXPR – funkcje przyjmujące dowolną liczbę argumentów przekazywanych przez stos
- ERRSET – funkcja pełniąca rolę try (w 1972 wprowadzono TRY i CATCH)
- Bignumy



# Co po LISPie 1 i 1.5?

InterLisp – tworzony przez BBN Technologies na PDP-10

# Co po LISPie 1 i 1.5?

InterLisp – tworzony przez BBN Technologies na PDP-10

- Wiele narzędzi programistycznych

# Co po LISPie 1 i 1.5?

InterLisp – tworzony przez BBN Technologies na PDP-10

- Wiele narzędzi programistycznych
  - ▶ *spelling corrector*

# Co po LISPie 1 i 1.5?

InterLisp – tworzony przez BBN Technologies na PDP-10

- Wiele narzędzi programistycznych
  - ▶ *spelling corrector*
  - ▶ „*Do what I mean*”

# Co po LISPie 1 i 1.5?

InterLisp – tworzony przez BBN Technologies na PDP-10

- Wiele narzędzi programistycznych
  - ▶ *spelling corrector*
  - ▶ „*Do what I mean*”
  - ▶ Masterscope – narzędzie do wyszukiwania informacji (np. która funkcja wywołuje którą) o funkcjach w dużych systemach.

# Co po LISPie 1 i 1.5?

InterLisp – tworzony przez BBN Technologies na PDP-10

- Wiele narzędzi programistycznych
  - ▶ *spelling corrector*
  - ▶ „*Do what I mean*”
  - ▶ Masterscope – narzędzie do wyszukiwania informacji (np. która funkcja wywołuje którą) o funkcjach w dużych systemach.
- Supernawiasy (nawias kwadratowy zamykał wszystkie otwarte nawiasy)

# Co po LISPie 1 i 1.5?

InterLisp – tworzony przez BBN Technologies na PDP-10

- Wiele narzędzi programistycznych
  - ▶ *spelling corrector*
  - ▶ „*Do what I mean*”
  - ▶ Masterscope – narzędzie do wyszukiwania informacji (np. która funkcja wywołuje którą) o funkcjach w dużych systemach.
- Supernawiasy (nawias kwadratowy zamykał wszystkie otwarte nawiasy)
- „*spaghetti stacks*”

