Prolog at 50

Manuel Hermenegildo[1,2]

[1]T. U. of Madrid (UPM)
[2]IMDEA Software Institute

*Part of the contents of this talk appear in the recent TPLP paper "***50 years of Prolog and Beyond***," by*

Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl,
Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz,
Salvador Abreu, and Giovanni Ciatto

written for Prolog's 50th anniversary and TPLP's 20th anniversary.

CILC'22 – Bologna, June 30, 2022

Prolog at 50 more?

Manuel Hermenegildo[1,2]

[1]T. U. of Madrid (UPM)
[2]IMDEA Software Institute

*Part of the contents of this talk appear in the recent TPLP paper "***50 years of Prolog and Beyond***," by*

Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl,
Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz,
Salvador Abreu, and Giovanni Ciatto

written for Prolog's 50th anniversary and TPLP's 20th anniversary.

CILC'22 – Bologna, June 30, 2022

The Year of Prolog

- Summer of 1972: Alain Colmerauer and his team in Marseille develop and implement the first version of Prolog.
- Together with both earlier and later collaborations with Bob Kowalski and colleagues in Edinburgh, this laid the practical and theoretical foundations for the Prolog and logic programming of today.
- We celebrate the 50th anniversary of these events through this
  "Year of Prolog"

  Organized by:
  The Association for Logic Programming and
  The Prolog Heritage Association
- Objectives:
  - Highlight the continuing significance of Prolog and LP for both symbolic, explainable AI, and computing more generally.
  - Inspire a new generation of students, by drawing their attention to the logic-based approach to computing.

The Year of Prolog

- Summer of 1972: Alain Colmerauer and his team in Marseille develop and implement the first version of Prolog.
- Together with both earlier and later collaborations with Bob Kowalski and colleagues in Edinburgh, this laid the practical and theoretical foundations for the Prolog and logic programming of today.
- We celebrate the 50th anniversary of these events through this

  "Year of Prolog"

  Organized by:
  The Association for Logic Programming and
  The Prolog Heritage Association
- Objectives:
  - Highlight the continuing significance of Prolog and LP for both symbolic, explainable AI, and computing more generally.
  - Inspire a new generation of students, by drawing their attention to the logic-based approach to computing.

The Year of Prolog

- Summer of 1972: Alain Colmerauer and his team in Marseille develop and implement the first version of Prolog.
- Together with both earlier and later collaborations with Bob Kowalski and colleagues in Edinburgh, this laid the practical and theoretical foundations for the Prolog and logic programming of today.

- We celebrate the 50th anniversary of these events through this

  "Year of Prolog"

  Organized by:
  The Association for Logic Programming and
  The Prolog Heritage Association

- Objectives:
  - Highlight the continuing significance of Prolog and LP for both symbolic, explainable AI, and computing more generally.
  - Inspire a new generation of students, by drawing their attention to the logic-based approach to computing.

The Year of Prolog

- Summer of 1972: Alain Colmerauer and his team in Marseille develop and implement the first version of Prolog.
- Together with both earlier and later collaborations with Bob Kowalski and colleagues in Edinburgh, this laid the practical and theoretical foundations for the Prolog and logic programming of today.

- We celebrate the 50th anniversary of these events through this
  "Year of Prolog"

  Organized by:
  The Association for Logic Programming and
  The Prolog Heritage Association

- Objectives:
  - Highlight the continuing significance of Prolog and LP for both symbolic, explainable AI, and computing more generally.
  - Inspire a new generation of students, by drawing their attention to the logic-based approach to computing.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.*
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., ICLP/FLoC, CILC!).

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.*
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., ICLP/FLoC, CILC!).

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.*
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., ICLP/FLoC, CILC!).

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.*
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., ICLP/FLoC, CILC!).

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.*
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., ICLP/FLoC, CILC!).

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.*
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., ICLP/FLoC, CILC!).

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.*
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., ICLP/FLoC, CILC!).

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

The Year of Prolog

- Initiatives:
  - **ALP Alain Colmerauer Prolog Heritage Prize**. *For recent practical accomplishments that highlight the benefits of Prolog-inspired computing for the future.*
  - **Prolog Day Symposium** (November 10, 2022) in which the Alain Colmerauer Prize will be awarded (subsequent editions at ICLP).
  - **Prolog Education initiative** (long-term initiative):
    - map and provide Prolog education resources for educators,
    - introduce schoolchildren/young adults to logic, programming, and AI w/Prolog.
  - **Survey paper on Fifty Years of Prolog and Beyond** published in the 20th anniversary special issue of TPLP.
  - **Special sessions and invited talks** at conferences (e.g., ICLP/FLoC, CILC!).

Activities are overseen by a Scientific Committee, chaired by Bob Kowalski.

- So, Prolog is 50!
  - What, 50 years?!? Half a century?!?!
  - Is Prolog therefore now 'old'? Is Prolog now irrelevant?

- Actually... continued, interest:
  - Many *active implementations*, and *more appearing* continuously.
  - TIOBE index of programming languages shows Prolog:
    - In upper 10% of all languages tracked (270).
    - Stable, even somewhat upward trend since 2012.
    - One of only 13 languages that are tracked 'long term'.
  - A truly impressive body of research and scientific firsts.

- So, Prolog is 50!
  - What, 50 years?!? Half a century?!?!
  - Is Prolog therefore now 'old'? Is Prolog now irrelevant?

- Actually... continued, interest:
  - Many *active implementations*, and *more appearing* continuously.
  - TIOBE index of programming languages shows Prolog:
    - In upper 10% of all languages tracked (270).
    - Stable, even somewhat upward trend since 2012.
    - One of only 13 languages that are tracked 'long term'.
  - A truly impressive body of research and scientific firsts.

- So, Prolog is 50!
  - What, 50 years?!? Half a century?!?!
  - Is Prolog therefore now 'old'? Is Prolog now irrelevant?

- Actually... continued, interest:
  - Many *active implementations*, and *more appearing* continuously.
  - TIOBE index of programming languages shows Prolog:
    - In upper 10% of all languages tracked (270).
    - Stable, even somewhat upward trend since 2012.
    - One of only 13 languages that are tracked 'long term'.
  - A truly impressive body of research and scientific firsts.

- So, Prolog is 50!
  - What, 50 years?!? Half a century?!?!
  - Is Prolog therefore now 'old'? Is Prolog now irrelevant?

- Actually... continued, interest:
  - Many *active implementations*, and *more appearing* continuously.
  - TIOBE index of programming languages shows Prolog:
    - In upper 10% of all languages tracked (270).
    - Stable, even somewhat upward trend since 2012.
    - One of only 13 languages that are tracked 'long term'.
  - A truly impressive body of research and scientific firsts.

Early steps, ancestries, and milestones

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.

  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).

  → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

  → Prolog! (1972 and 1973)

  - Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).

  → Dec-10 Prolog

... ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972 and 1973)

  - Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).
  - → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).

  → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  → Prolog! (1972 and 1973)

  - Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).

  → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).

  → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

  → Prolog! (1972 and 1973)

  - Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).

  → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).

→ Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

→ Prolog! (1972 and 1973)

- Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).

→ Dec-10 Prolog

. . . ⇒

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).

  → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

  → Prolog! (1972 and 1973)

  - Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).

  → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).

  → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

  → Prolog! (1972 and 1973)

  - Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).

  → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).

  → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

  → Prolog! (1972 and 1973)

  - Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).

  → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).

→ Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

→ Prolog! (1972 and 1973)

- Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).

→ Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.
- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).

  → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

  → Prolog! (1972 and 1973)

  - Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).

  → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972 and 1973)

  - Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).
  - → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).

  → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

  → Prolog! (1972 and 1973)

  - Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).

  → Dec-10 Prolog

... ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).

  → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).

  → Prolog! (1972 and 1973)

  - Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).

  → Dec-10 Prolog

. . . ⟹

Ancestors and birth

- Not possible to do full justice in this talk!
  - See Kowalski (1988,2013), Cohen (1988), VanRoy (1994), Colmerauer (1996), Gupta et al. (2001), vanEmden (2006), McJones's archive, etc.

- Anyway, some highlights:
  - The AI language LISP (McCarthy 1962) → "very high-level languages."
  - Resolution inference rule (Robinson 1965).
    - Semi-decision procedure for first-order (FO-)logic.
  - Boyer and Moore: structure sharing.
  - Cordell Green (1969): extend resolution to automatically answer questions in FO-logic (a high point for Logic in AI).
  - Ted Elcock: Aberdeen System ("Absys").
  - Colmerauer: Q-systems (1970).
  - Kowalski and Kuehner: SL-resolution (1971).
  - → Marseilles - Edinburgh collaboration (Colmerauer/Kowalski and teams).
  - → Prolog! (1972 and 1973)

  - Edinburgh, + Lisbon collaboration (DHD Warren, Pereira(s), Bowen, Byrd).
  - → Dec-10 Prolog

. . . ⇒

Approximate timeline of some early Prologs (up to ISO)

1972 Prolog 0

1973 Prolog I

1975 CDL-Prolog

1975 DEC-10 Prolog

1982 Prolog II

1982 C-Prolog, MU-Prolog

1983 **(WAM)**

1985 Quintus

1986 SICStus, &-Prolog (Ciao)

1986 CLP($\mathcal{R}$)

1986 SWI-Prolog

1987 SB Prolog

1988 CHIP

1992 wamcc

1992 BinProlog

1993 ECL^iPS^e

1993 Ciao Prolog

1994 XSB

1994 B-Prolog

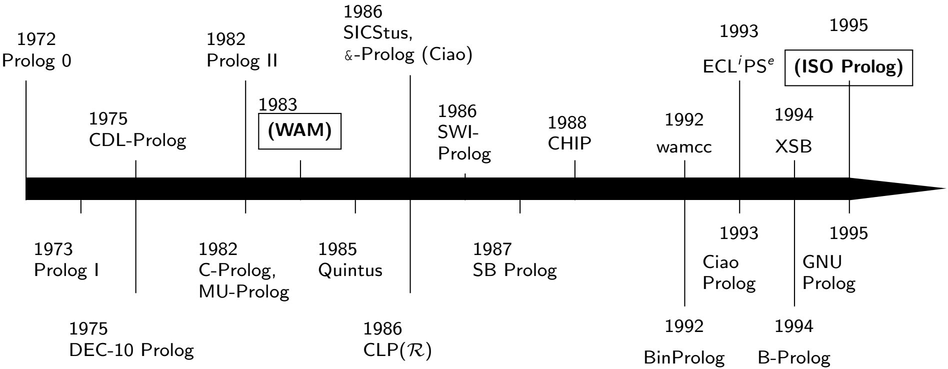
1995 **(ISO Prolog)**

1995 GNU Prolog

Plus several others: Unix Prolog, Waterloo Prolog, UNSW Prolog, ...
Some early systems may not fit a modern definition of Prolog.
All this progressed in parallel with further advances in the theoretical underpinnings:

- Kowalski/van Emden (1976) linear res. for Horn clauses, no factoring rule, ...
- Clark (1978) correctness of NaF w.r.t. program completion.
- Reiter (1978) formalization of "Closed world assumption."
- Minker, Gallaire, Cohen, Lassez/Jaffar/Maher, DHD Warren, Tamaki/Sato, DS Warren, ...

Approximate timeline of some early Prologs (up to ISO)



1972
Prolog 0

1973
Prolog I

1975
CDL-Prolog

1975
DEC-10 Prolog

1982
Prolog II

1982
C-Prolog,
MU-Prolog

1983
(WAM)

1985
Quintus

1986
SICStus,
&-Prolog (Ciao)

1986
CLP($\mathcal{R}$)

1986
SWI-Prolog

1987
SB Prolog

1988
CHIP

1992
wamcc

1992
BinProlog

1993
ECL^{i}PS^e

1993
Ciao Prolog

1994
XSB

1994
B-Prolog

1995
(ISO Prolog)

1995
GNU Prolog

Plus several others: Unix Prolog, Waterloo Prolog, UNSW Prolog, ...

Some early systems may not fit a modern definition of Prolog.

All this progressed in parallel with further advances in the theoretical underpinnings:

- Kowalski/van Emden (1976) linear res. for Horn clauses, no factoring rule, ...
- Clark (1978) correctness of NaF w.r.t. program completion.
- Reiter (1978) formalization of "Closed world assumption."
- Minker, Gallaire, Cohen, Lassez/Jaffar/Maher, DHD Warren, Tamaki/Sato, DS Warren, ...

Approximate timeline of some early Prologs (up to ISO)

1972 Prolog 0

1973 Prolog I

1975 CDL-Prolog

1975 DEC-10 Prolog

1982 Prolog II

1982 C-Prolog, MU-Prolog

1983 **(WAM)**

1985 Quintus

1986 SICStus, &-Prolog (Ciao)

1986 CLP($\mathcal{R}$)

1986 SWI-Prolog

1987 SB Prolog

1988 CHIP

1992 wamcc

1992 BinProlog

1993 ECL^{i}PS^e

1993 Ciao Prolog

1994 XSB

1994 B-Prolog

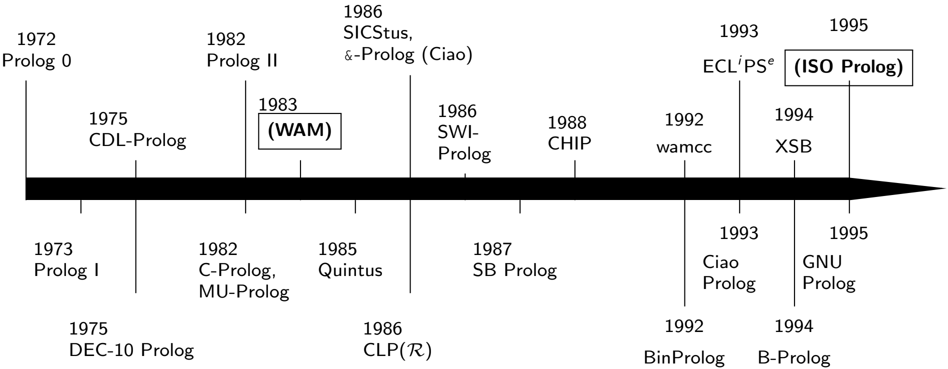
1995 **(ISO Prolog)**

1995 GNU Prolog

Plus several others: Unix Prolog, Waterloo Prolog, UNSW Prolog, ...
Some early systems may not fit a modern definition of Prolog.
All this progressed in parallel with further advances in the theoretical underpinnings:

- Kowalski/van Emden (1976) linear res. for Horn clauses, no factoring rule, ...
- Clark (1978) correctness of NaF w.r.t. program completion.
- Reiter (1978) formalization of "Closed world assumption."
- Minker, Gallaire, Cohen, Lassez/Jaffar/Maher, DHD Warren, Tamaki/Sato, DS Warren, ...

Approximate timeline of some early Prologs (up to ISO)



1972 Prolog 0

1973 Prolog I

1975 CDL-Prolog

1975 DEC-10 Prolog

1982 Prolog II

1982 C-Prolog, MU-Prolog

1983 **(WAM)**

1985 Quintus

1986 SICStus, &-Prolog (Ciao)

1986 CLP($\mathcal{R}$)

1986 SWI-Prolog

1987 SB Prolog

1988 CHIP

1992 wamcc

1992 BinProlog

1993 $\text{ECL}^{i}\text{PS}^{e}$

1993 Ciao Prolog

1994 XSB

1994 B-Prolog

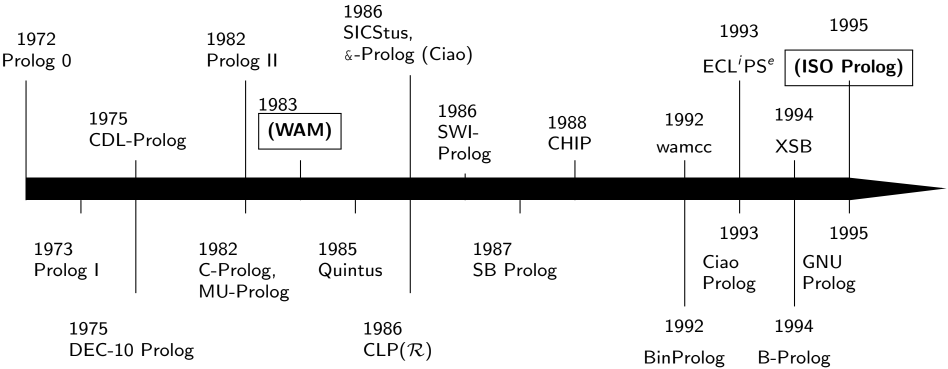
1995 **(ISO Prolog)**

1995 GNU Prolog

Plus several others: Unix Prolog, Waterloo Prolog, UNSW Prolog, ...
Some early systems may not fit a modern definition of Prolog.
All this progressed in parallel with further advances in the theoretical underpinnings:

- Kowalski/van Emden (1976) linear res. for Horn clauses, no factoring rule, ...
- Clark (1978) correctness of NaF w.r.t. program completion.
- Reiter (1978) formalization of "Closed world assumption."
- Minker, Gallaire, Cohen, Lassez/Jaffar/Maher, DHD Warren, Tamaki/Sato, DS Warren, ...

Approximate timeline of some early Prologs (up to ISO)



1972 Prolog 0

1973 Prolog I

1975 CDL-Prolog

1975 DEC-10 Prolog

1982 Prolog II

1982 C-Prolog, MU-Prolog

1983 **(WAM)**

1985 Quintus

1986 SICStus, &-Prolog (Ciao)

1986 SWI-Prolog

1986 CLP($\mathcal{R}$)

1987 SB Prolog

1988 CHIP

1992 wamcc

1992 BinProlog

1993 ECL^iPS^e

1993 Ciao Prolog

1994 XSB

1994 B-Prolog

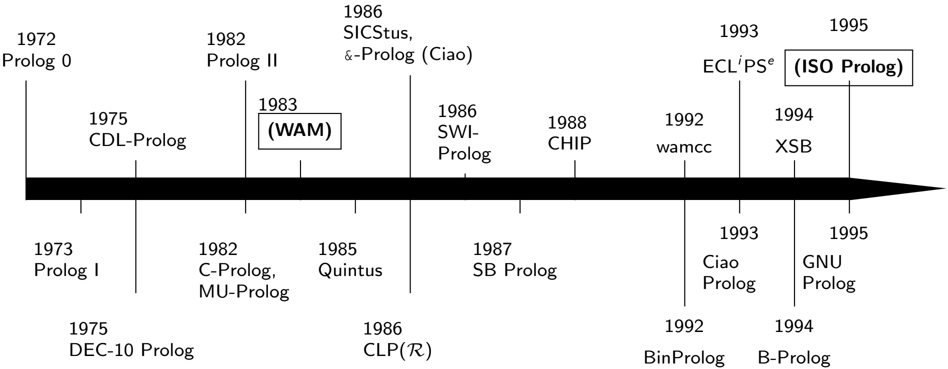
1995 **(ISO Prolog)**

1995 GNU Prolog

Plus several others: Unix Prolog, Waterloo Prolog, UNSW Prolog, ...
Some early systems may not fit a modern definition of Prolog.
All this progressed in parallel with further advances in the theoretical underpinnings:

- Kowalski/van Emden (1976) linear res. for Horn clauses, no factoring rule, ...
- Clark (1978) correctness of NaF w.r.t. program completion.
- Reiter (1978) formalization of "Closed world assumption."
- Minker, Gallaire, Cohen, Lassez/Jaffar/Maher, DHD Warren, Tamaki/Sato, DS Warren, ...

Approximate timeline of some early Prologs (up to ISO)



1972 Prolog 0

1973 Prolog I

1975 CDL-Prolog

1975 DEC-10 Prolog

1982 Prolog II

1982 C-Prolog, MU-Prolog

1983 **(WAM)**

1985 Quintus

1986 SICStus, &-Prolog (Ciao)

1986 CLP($\mathcal{R}$)

1986 SWI-Prolog

1987 SB Prolog

1988 CHIP

1992 wamcc

1992 BinProlog

1993 ECL^iPS^e

1993 Ciao Prolog

1994 XSB

1994 B-Prolog

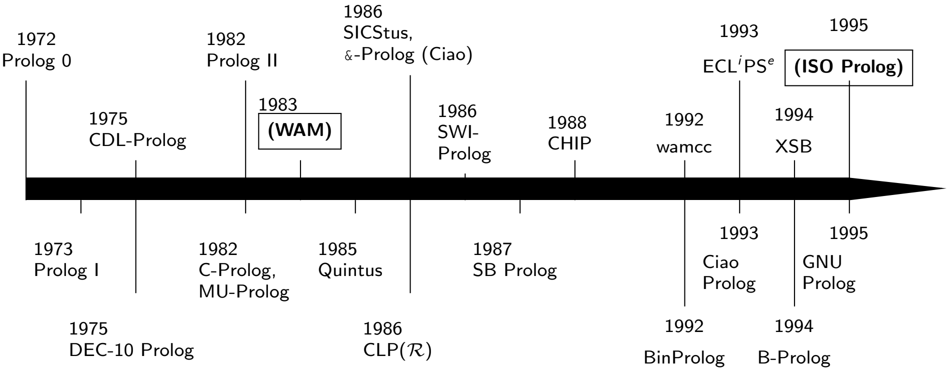
1995 **(ISO Prolog)**

1995 GNU Prolog

Plus several others: Unix Prolog, Waterloo Prolog, UNSW Prolog, ...
Some early systems may not fit a modern definition of Prolog.
All this progressed in parallel with further advances in the theoretical underpinnings:

- Kowalski/van Emden (1976) linear res. for Horn clauses, no factoring rule, ...
- Clark (1978) correctness of NaF w.r.t. program completion.
- Reiter (1978) formalization of "Closed world assumption."
- Minker, Gallaire, Cohen, Lassez/Jaffar/Maher, DHD Warren, Tamaki/Sato, DS Warren, ...

Approximate timeline of some early Prologs (up to ISO)



1972
Prolog 0

1973
Prolog I

1975
CDL-Prolog

1975
DEC-10 Prolog

1982
Prolog II

1982
C-Prolog,
MU-Prolog

1983
(WAM)

1985
Quintus

1986
SICStus,
&-Prolog (Ciao)

1986
CLP($\mathcal{R}$)

1986
SWI-Prolog

1987
SB Prolog

1988
CHIP

1992
wamcc

1992
BinProlog

1993
ECL^{i}PS^e

1993
Ciao Prolog

1994
XSB

1994
B-Prolog

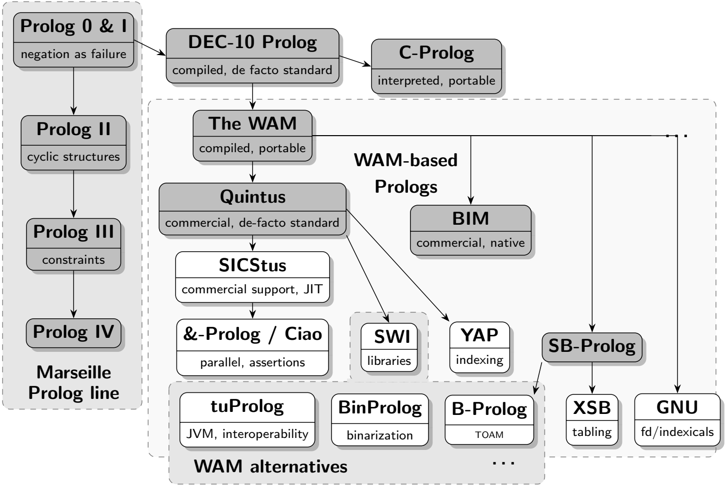
1995
(ISO Prolog)

1995
GNU Prolog

Plus several others: Unix Prolog, Waterloo Prolog, UNSW Prolog, ...
Some early systems may not fit a modern definition of Prolog.
All this progressed in parallel with further advances in the theoretical underpinnings:

- Kowalski/van Emden (1976) linear res. for Horn clauses, no factoring rule, ...
- Clark (1978) correctness of NaF w.r.t. program completion.
- Reiter (1978) formalization of "Closed world assumption."
- Minker, Gallaire, Cohen, Lassez/Jaffar/Maher, DHD Warren, Tamaki/Sato, DS Warren, ...

Prolog system heritage



Prolog 0 & I
negation as failure

Prolog II
cyclic structures

Prolog III
constraints

Prolog IV

Marseille Prolog line

DEC-10 Prolog
compiled, de facto standard

C-Prolog
interpreted, portable

The WAM
compiled, portable

WAM-based Prologs

Quintus
commercial, de-facto standard

BIM
commercial, native

SICStus
commercial support, JIT

&-Prolog / Ciao
parallel, assertions

SWI
libraries

YAP
indexing

SB-Prolog

tuProlog
JVM, interoperability

BinProlog
binarization

B-Prolog
TOAM

XSB
tabling

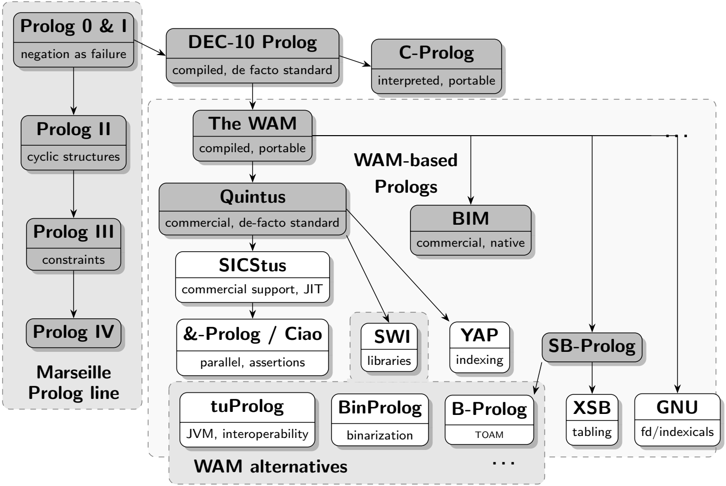
GNU
fd/indexicals

WAM alternatives . . .

Arrows:	influences and inspiration.
White background:	currently active/supported systems.
Lower legends:	just some highlight(s) (see later).

Again, more missing!: ECL^{i}PS^e, IBM, LIFE, Andorra-I, Scryer, Tau, ...

Prolog system heritage



Prolog 0 & I

negation as failure

DEC-10 Prolog

compiled, de facto standard

C-Prolog

interpreted, portable

Prolog II

cyclic structures

The WAM

compiled, portable

WAM-based Prologs

Quintus

commercial, de-facto standard

BIM

commercial, native

Prolog III

constraints

SICStus

commercial support, JIT

Prolog IV

&-Prolog / Ciao

parallel, assertions

SWI

libraries

YAP

indexing

SB-Prolog

Marseille Prolog line

tuProlog

JVM, interoperability

BinProlog

binarization

B-Prolog

TOAM

XSB

tabling

GNU

fd/indexicals

WAM alternatives

...

...

Arrows:	influences and inspiration.
White background:	currently active/supported systems.
Lower legends:	just some highlight(s) (see later).

Again, more missing!: ECL^{i}PS^e, IBM, LIFE, Andorra-I, Scryer, Tau, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
  - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, “opening the box”, CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
  - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, "opening the box", CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
    - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, "opening the box", CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
  - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, “opening the box”, CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
  - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, "opening the box", CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
  - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, "opening the box", CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
    - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, "opening the box", CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
  - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, “opening the box”, CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
    - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, "opening the box", CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
    - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, "opening the box", CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
  - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, "opening the box", CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
    - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, “opening the box”, CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
  - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, "opening the box", CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

Some major milestones

- Of course, the first Prolog(s).
- Dec-10 Prolog: Compilation (+ improved syntax, etc.)

→ the WAM.

- Optimization (SICStus, VanRoy, ...), GC, Parallelism.
  - Analysis (abstract interpretation).

→ Performance!

- FGCS → MCC → ECRC → ESPRIT → EU research programs (and others)

- Constraints (CLP scheme/CLP(R), FD, "opening the box", CHR).
- Early ded., Tabling, SLG-resolution, minimal-model / well-founded semantics.
- Higher-order, functional syntax.
- Types/modes, properties.
- Learning (ILP), probabilistic.
- Constructive negation, ASP, s(CASP).

- Applications of techniques to other languages, combination with Neural Networks, Explainable AI, ...

An overview of current systems

Current systems: focus, some highlights

System	Some highlights
B-Prolog	Action rules, efficient CLP supporting many data structures.
Ciao	Multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (incl. types, modes), performance/scalabilty, language interfaces, HO, parallelism.
ECL^{i}PS^e	Focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog.
GNU	Extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs.
JIProlog	Interpreter in Java, embeddable, semantic intelligence / NLP applications.
Scryer	New Prolog in development, aims at full ISO conformance.
SICStus	Commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation.
SWI	General-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL^{i}PS^e and XSB.
tuProlog	Bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library.
XSB	Commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming).
YAP	Focus on scalability, advanced indexing, language integrations (Python, R), integration of databases.

Current systems: focus, some highlights

System	Some highlights
B-Prolog	Action rules, efficient CLP supporting many data structures.
Ciao	Multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (incl. types, modes), performance/scalabilty, language interfaces, HO, parallelism.
ECL^{i}PS^e	Focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog.
GNU	Extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs.
JIProlog	Interpreter in Java, embeddable, semantic intelligence / NLP applications.
Scryer	New Prolog in development, aims at full ISO conformance.
SICStus	Commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation.
SWI	General-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL^{i}PS^e and XSB.
tuProlog	Bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library.
XSB	Commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming).
YAP	Focus on scalability, advanced indexing, language integrations (Python, R), integration of databases.

Current systems: focus, some highlights

System	Some highlights
B-Prolog	Action rules, efficient CLP supporting many data structures.
Ciao	Multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (incl. types, modes), performance/scalabilty, language interfaces, HO, parallelism.
ECL^{i}PS^e	Focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog.
GNU	Extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs.
JIProlog	Interpreter in Java, embeddable, semantic intelligence / NLP applications.
Scryer	New Prolog in development, aims at full ISO conformance.
SICStus	Commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation.
SWI	General-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL^{i}PS^e and XSB.
tuProlog	Bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library.
XSB	Commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming).
YAP	Focus on scalability, advanced indexing, language integrations (Python, R), integration of databases.

Current systems: focus, some highlights

System	Some highlights
B-Prolog	Action rules, efficient CLP supporting many data structures.
Ciao	Multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (incl. types, modes), performance/scalabilty, language interfaces, HO, parallelism.
ECL^{i}PS^e	Focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog.
GNU	Extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs.
JIProlog	Interpreter in Java, embeddable, semantic intelligence / NLP applications.
Scryer	New Prolog in development, aims at full ISO conformance.
SICStus	Commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation.
SWI	General-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL^{i}PS^e and XSB.
tuProlog	Bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library.
XSB	Commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming).
YAP	Focus on scalability, advanced indexing, language integrations (Python, R), integration of databases.

Current systems: focus, some highlights

System	Some highlights
B-Prolog	Action rules, efficient CLP supporting many data structures.
Ciao	Multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (incl. types, modes), performance/scalabilty, language interfaces, HO, parallelism.
ECL^{i}PS^e	Focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog.
GNU	Extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs.
JIProlog	Interpreter in Java, embeddable, semantic intelligence / NLP applications.
Scryer	New Prolog in development, aims at full ISO conformance.
SICStus	Commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation.
SWI	General-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL^{i}PS^e and XSB.
tuProlog	Bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library.
XSB	Commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming).
YAP	Focus on scalability, advanced indexing, language integrations (Python, R), integration of databases.

Current systems: focus, some highlights

System	Some highlights
B-Prolog	Action rules, efficient CLP supporting many data structures.
Ciao	Multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (incl. types, modes), performance/scalabilty, language interfaces, HO, parallelism.
ECL^{i}PS^e	Focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog.
GNU	Extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs.
JIProlog	Interpreter in Java, embeddable, semantic intelligence / NLP applications.
Scryer	New Prolog in development, aims at full ISO conformance.
SICStus	Commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation.
SWI	General-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL^{i}PS^e and XSB.
tuProlog	Bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library.
XSB	Commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming).
YAP	Focus on scalability, advanced indexing, language integrations (Python, R), integration of databases.

Current systems: focus, some highlights

System	Some highlights
B-Prolog	Action rules, efficient CLP supporting many data structures.
Ciao	Multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (incl. types, modes), performance/scalabilty, language interfaces, HO, parallelism.
ECL^{i}PS^e	Focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog.
GNU	Extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs.
JIProlog	Interpreter in Java, embeddable, semantic intelligence / NLP applications.
Scryer	New Prolog in development, aims at full ISO conformance.
SICStus	Commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation.
SWI	General-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL^{i}PS^e and XSB.
tuProlog	Bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library.
XSB	Commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming).
YAP	Focus on scalability, advanced indexing, language integrations (Python, R), integration of databases.

Current systems: focus, some highlights

System	Some highlights
B-Prolog	Action rules, efficient CLP supporting many data structures.
Ciao	Multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (incl. types, modes), performance/scalabilty, language interfaces, HO, parallelism.
ECL^{i}PS^e	Focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog.
GNU	Extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs.
JIProlog	Interpreter in Java, embeddable, semantic intelligence / NLP applications.
Scryer	New Prolog in development, aims at full ISO conformance.
SICStus	Commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation.
SWI	General-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL^{i}PS^e and XSB.
tuProlog	Bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library.
XSB	Commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming).
YAP	Focus on scalability, advanced indexing, language integrations (Python, R), integration of databases.

Current systems: focus, some highlights

System	Some highlights
B-Prolog	Action rules, efficient CLP supporting many data structures.
Ciao	Multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (incl. types, modes), performance/scalabilty, language interfaces, HO, parallelism.
ECL^{i}PS^e	Focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog.
GNU	Extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs.
JIProlog	Interpreter in Java, embeddable, semantic intelligence / NLP applications.
Scryer	New Prolog in development, aims at full ISO conformance.
SICStus	Commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation.
SWI	General-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL^{i}PS^e and XSB.
tuProlog	Bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library.
XSB	Commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming).
YAP	Focus on scalability, advanced indexing, language integrations (Python, R), integration of databases.

Current systems: focus, some highlights

System	Some highlights
B-Prolog	Action rules, efficient CLP supporting many data structures.
Ciao	Multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (incl. types, modes), performance/scalabilty, language interfaces, HO, parallelism.
ECL^{i}PS^e	Focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog.
GNU	Extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs.
JIProlog	Interpreter in Java, embeddable, semantic intelligence / NLP applications.
Scryer	New Prolog in development, aims at full ISO conformance.
SICStus	Commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation.
SWI	General-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL^{i}PS^e and XSB.
tuProlog	Bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library.
XSB	Commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming).
YAP	Focus on scalability, advanced indexing, language integrations (Python, R), integration of databases.

Current systems: focus, some highlights

System	Some highlights
B-Prolog	Action rules, efficient CLP supporting many data structures.
Ciao	Multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (incl. types, modes), performance/scalabilty, language interfaces, HO, parallelism.
ECL^{i}PS^e	Focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog.
GNU	Extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs.
JIProlog	Interpreter in Java, embeddable, semantic intelligence / NLP applications.
Scryer	New Prolog in development, aims at full ISO conformance.
SICStus	Commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation.
SWI	General-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL^{i}PS^e and XSB.
tuProlog	Bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library.
XSB	Commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming).
YAP	Focus on scalability, advanced indexing, language integrations (Python, R), integration of databases.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
    - Module system
    - Built-in data types
    - Foreign language interface
- Libraries and extensions:
    - Constraints/CHR
    - Data structures
    - Tabling
    - Parallelism
    - Indexing
    - Type and modes
    - Coroutining
    - Testing
    - Debugging
    - Mutable terms
- Tools:
    - Debugging
    - Unit testing
    - Property Verification (incl. types and modes)
    - Auto-documentation
    - Playgrounds, etc.

A selection of features

- Core features:
    - Module system
    - Built-in data types
    - Foreign language interface
- Libraries and extensions:
    - Constraints/CHR
    - Data structures
    - Tabling
    - Parallelism
    - Indexing
    - Type and modes
    - Coroutining
    - Testing
    - Debugging
    - Mutable terms
- Tools:
    - Debugging
    - Unit testing
    - Property Verification (incl. types and modes)
    - Auto-documentation
    - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
    - Module system
    - Built-in data types
    - Foreign language interface
- Libraries and extensions:
    - Constraints/CHR
    - Data structures
    - Tabling
    - Parallelism
    - Indexing
    - Type and modes
    - Coroutining
    - Testing
    - Debugging
    - Mutable terms
- Tools:
    - Debugging
    - Unit testing
    - Property Verification (incl. types and modes)
    - Auto-documentation
    - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
    - Module system
    - Built-in data types
    - Foreign language interface
- Libraries and extensions:
    - Constraints/CHR
    - Data structures
    - Tabling
    - Parallelism
    - Indexing
    - Type and modes
    - Coroutining
    - Testing
    - Debugging
    - Mutable terms
- Tools:
    - Debugging
    - Unit testing
    - Property Verification (incl. types and modes)
    - Auto-documentation
    - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

A selection of features

- Core features:
  - Module system
  - Built-in data types
  - Foreign language interface
- Libraries and extensions:
  - Constraints/CHR
  - Data structures
  - Tabling
  - Parallelism
  - Indexing
  - Type and modes
  - Coroutining
  - Testing
  - Debugging
  - Mutable terms
- Tools:
  - Debugging
  - Unit testing
  - Property Verification (incl. types and modes)
  - Auto-documentation
  - Playgrounds, etc.

Support status for selected features - I

System	Open Src.	Modules	Non-Std. Data Types	Foreign Language Interfaces
B-Prolog	✗	✗	arrays, sets, hashtables	C, Java
Ciao	✓	✓	✗	C, Java, Python, JScrpt
ECLiPSe	✓	✓	arrays, strings	C, Java, Python, PHP
GNU Prolog	✓	✗	arrays	C, Java, PHP
JIProlog	✓	✓	✗	Java
SICStus	✗	✓	✗	C, Java, .NET, Tcl/Tk
SWI	✓	✓	dicts, strings	C, C++, Java
τProlog	✓	✓	✗	JavaScript
tuProlog	✓	✗	arrays	Java, .NET, Android, iOS
XSB	✓	✓	✗	C, Java, PERL
YAP	✓	✓	✗	C, Python, R

Support status for selected features - I

System	Open Src.	Modules	Non-Std. Data Types	Foreign Language Interfaces
B-Prolog	✗	✗	arrays, sets, hashtables	C, Java
Ciao	✓	✓	✗	C, Java, Python, JScrpt
ECLiPSe	✓	✓	arrays, strings	C, Java, Python, PHP
GNU Prolog	✓	✗	arrays	C, Java, PHP
JIProlog	✓	✓	✗	Java
SICStus	✗	✓	✗	C, Java, .NET, Tcl/Tk
SWI	✓	✓	dicts, strings	C, C++, Java
τProlog	✓	✓	✗	JavaScript
tuProlog	✓	✗	arrays	Java, .NET, Android, iOS
XSB	✓	✓	✗	C, Java, PERL
YAP	✓	✓	✗	C, Python, R

Support status for selected features - I

System	Open Src.	Modules	Non-Std. Data Types	Foreign Language Interfaces
B-Prolog	✗	✗	arrays, sets, hashtables	C, Java
Ciao	✓	✓	✗	C, Java, Python, JScrpt
ECLiPSe	✓	✓	arrays, strings	C, Java, Python, PHP
GNU Prolog	✓	✗	arrays	C, Java, PHP
JIProlog	✓	✓	✗	Java
SICStus	✗	✓	✗	C, Java, .NET, Tcl/Tk
SWI	✓	✓	dicts, strings	C, C++, Java
τProlog	✓	✓	✗	JavaScript
tuProlog	✓	✗	arrays	Java, .NET, Android, iOS
XSB	✓	✓	✗	C, Java, PERL
YAP	✓	✓	✗	C, Python, R

Support status for selected features - I

System	Open Src.	Modules	Non-Std. Data Types	Foreign Language Interfaces
B-Prolog	✗	✗	arrays, sets, hashtables	C, Java
Ciao	✓	✓	✗	C, Java, Python, JScrpt
ECLiPSe	✓	✓	arrays, strings	C, Java, Python, PHP
GNU Prolog	✓	✗	arrays	C, Java, PHP
JIProlog	✓	✓	✗	Java
SICStus	✗	✓	✗	C, Java, .NET, Tcl/Tk
SWI	✓	✓	dicts, strings	C, C++, Java
τProlog	✓	✓	✗	JavaScript
tuProlog	✓	✗	arrays	Java, .NET, Android, iOS
XSB	✓	✓	✗	C, Java, PERL
YAP	✓	✓	✗	C, Python, R

Support status for selected features - I

System	Open Src.	Modules	Non-Std. Data Types	Foreign Language Interfaces
B-Prolog	✗	✗	arrays, sets, hashtables	C, Java
Ciao	✓	✓	✗	C, Java, Python, JScrpt
ECLiPSe	✓	✓	arrays, strings	C, Java, Python, PHP
GNU Prolog	✓	✗	arrays	C, Java, PHP
JIProlog	✓	✓	✗	Java
SICStus	✗	✓	✗	C, Java, .NET, Tcl/Tk
SWI	✓	✓	dicts, strings	C, C++, Java
τProlog	✓	✓	✗	JavaScript
tuProlog	✓	✗	arrays	Java, .NET, Android, iOS
XSB	✓	✓	✗	C, Java, PERL
YAP	✓	✓	✗	C, Python, R

Support status for selected features - II

System	CLP	CHR	Tabling	Parallelism	Indexing	Types/Modes
B-Prolog	$\mathcal{FD}$, $\mathcal{B}$, *Set*	✓	✓	✗	N-FA	✗
Ciao	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	FA, MA	✓
ECLiPSe	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, *Set*	✓	✗	✓	most suitable	✗
GNU Prolog	$\mathcal{FD}$, $\mathcal{B}$	✗	✗	✗	FA	✗
JIProlog	✗	✗	✗	✗	undocumented	✗
SICStus	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✗	✗	FA	✗
SWI	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	MA, deep, JIT	✗
τProlog	✗	✗	✗	✗	undocumented	✗
tuProlog	✗	✗	✗	✓	FA	✗
XSB	$\mathcal{R}$	✓	✓	✓	all, trie	✗
YAP	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✗	FA, MA, JIT	✗

Support status for selected features - II

System	CLP	CHR	Tabling	Parallelism	Indexing	Types/Modes
B-Prolog	$\mathcal{FD}$, $\mathcal{B}$, Set	✓	✓	✗	N-FA	✗
Ciao	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	FA, MA	✓
ECLiPSe	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, Set	✓	✗	✓	most suitable	✗
GNU Prolog	$\mathcal{FD}$, $\mathcal{B}$	✗	✗	✗	FA	✗
JIProlog	✗	✗	✗	✗	undocumented	✗
SICStus	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✗	✗	FA	✗
SWI	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	MA, deep, JIT	✗
τProlog	✗	✗	✗	✗	undocumented	✗
tuProlog	✗	✗	✗	✓	FA	✗
XSB	$\mathcal{R}$	✓	✓	✓	all, trie	✗
YAP	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✗	FA, MA, JIT	✗

Support status for selected features - II

System	CLP	CHR	Tabling	Parallelism	Indexing	Types/Modes
B-Prolog	$\mathcal{FD}$, $\mathcal{B}$, *Set*	✓	✓	✗	N-FA	✗
Ciao	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	FA, MA	✓
ECLiPSe	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, *Set*	✓	✗	✓	most suitable	✗
GNU Prolog	$\mathcal{FD}$, $\mathcal{B}$	✗	✗	✗	FA	✗
JIProlog	✗	✗	✗	✗	undocumented	✗
SICStus	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✗	✗	FA	✗
SWI	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	MA, deep, JIT	✗
τProlog	✗	✗	✗	✗	undocumented	✗
tuProlog	✗	✗	✗	✓	FA	✗
XSB	$\mathcal{R}$	✓	✓	✓	all, trie	✗
YAP	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✗	FA, MA, JIT	✗

Support status for selected features - II

System	CLP	CHR	Tabling	Parallelism	Indexing	Types/Modes
B-Prolog	$\mathcal{FD}$, $\mathcal{B}$, *Set*	✓	✓	✗	N-FA	✗
Ciao	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	FA, MA	✓
ECLiPSe	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, *Set*	✓	✗	✓	most suitable	✗
GNU Prolog	$\mathcal{FD}$, $\mathcal{B}$	✗	✗	✗	FA	✗
JIProlog	✗	✗	✗	✗	undocumented	✗
SICStus	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✗	✗	FA	✗
SWI	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	MA, deep, JIT	✗
τProlog	✗	✗	✗	✗	undocumented	✗
tuProlog	✗	✗	✗	✓	FA	✗
XSB	$\mathcal{R}$	✓	✓	✓	all, trie	✗
YAP	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✗	FA, MA, JIT	✗

Support status for selected features - II

System	CLP	CHR	Tabling	Parallelism	Indexing	Types/Modes
B-Prolog	$\mathcal{FD}$, $\mathcal{B}$, *Set*	✓	✓	✗	N-FA	✗
Ciao	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	FA, MA	✓
ECLiPSe	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, *Set*	✓	✗	✓	most suitable	✗
GNU Prolog	$\mathcal{FD}$, $\mathcal{B}$	✗	✗	✗	FA	✗
JIProlog	✗	✗	✗	✗	undocumented	✗
SICStus	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✗	✗	FA	✗
SWI	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	MA, deep, JIT	✗
τProlog	✗	✗	✗	✗	undocumented	✗
tuProlog	✗	✗	✗	✓	FA	✗
XSB	$\mathcal{R}$	✓	✓	✓	all, trie	✗
YAP	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✗	FA, MA, JIT	✗

Support status for selected features - II

System	CLP	CHR	Tabling	Parallelism	Indexing	Types/Modes
B-Prolog	$\mathcal{FD}$, $\mathcal{B}$, *Set*	✓	✓	✗	N-FA	✗
Ciao	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	FA, MA	✓
ECLiPSe	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$, *Set*	✓	✗	✓	most suitable	✗
GNU Prolog	$\mathcal{FD}$, $\mathcal{B}$	✗	✗	✗	FA	✗
JIProlog	✗	✗	✗	✗	undocumented	✗
SICStus	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✗	✗	FA	✗
SWI	$\mathcal{FD}$, $\mathcal{B}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✓	MA, deep, JIT	✗
τProlog	✗	✗	✗	✗	undocumented	✗
tuProlog	✗	✗	✗	✓	FA	✗
XSB	$\mathcal{R}$	✓	✓	✓	all, trie	✗
YAP	$\mathcal{FD}$, $\mathcal{Q}$, $\mathcal{R}$	✓	✓	✗	FA, MA, JIT	✗

Support status for selected features - III

System	Coroutines	Testing	Debugger	Global Variables	Mutable Terms
B-Prolog	✓	✗	trace	✓	✗
Ciao	✓	✓	trace / source	✓	✓
ECLiPSe	✓	✓	trace	✓	✗
GNU Prolog	✗	✗	trace	✓	✓
JIProlog	✗	✗	trace	✗	✗
SICStus	✓	✓	trace / source	✗	✓
SWI	✓	✓	trace / graphical	✓	✓
τProlog	✗	✗	✗	✗	✗
tuProlog	✗	✗	spy	✗	✗
XSB	✓	✗	trace	✗	✗
YAP	✗	✗	trace	✓	✗

Support status for selected features - III

System	Coroutines	Testing	Debugger	Global Variables	Mutable Terms
B-Prolog	✓	✗	trace	✓	✗
Ciao	✓	✓	trace / source	✓	✓
ECLiPSe	✓	✓	trace	✓	✗
GNU Prolog	✗	✗	trace	✓	✓
JIProlog	✗	✗	trace	✗	✗
SICStus	✓	✓	trace / source	✗	✓
SWI	✓	✓	trace / graphical	✓	✓
τProlog	✗	✗	✗	✗	✗
tuProlog	✗	✗	spy	✗	✗
XSB	✓	✗	trace	✗	✗
YAP	✗	✗	trace	✓	✗

Support status for selected features - III

System	Coroutines	Testing	Debugger	Global Variables	Mutable Terms
B-Prolog	✓	✗	trace	✓	✗
Ciao	✓	✓	trace / source	✓	✓
ECLiPSe	✓	✓	trace	✓	✗
GNU Prolog	✗	✗	trace	✓	✓
JIProlog	✗	✗	trace	✗	✗
SICStus	✓	✓	trace / source	✗	✓
SWI	✓	✓	trace / graphical	✓	✓
τProlog	✗	✗	✗	✗	✗
tuProlog	✗	✗	spy	✗	✗
XSB	✓	✗	trace	✗	✗
YAP	✗	✗	trace	✓	✗

Support status for selected features - III

System	Coroutines	Testing	Debugger	Global Variables	Mutable Terms
B-Prolog	✓	✗	trace	✓	✗
Ciao	✓	✓	trace / source	✓	✓
ECLiPSe	✓	✓	trace	✓	✗
GNU Prolog	✗	✗	trace	✓	✓
JIProlog	✗	✗	trace	✗	✗
SICStus	✓	✓	trace / source	✗	✓
SWI	✓	✓	trace / graphical	✓	✓
τProlog	✗	✗	✗	✗	✗
tuProlog	✗	✗	spy	✗	✗
XSB	✓	✗	trace	✗	✗
YAP	✗	✗	trace	✓	✗

Support status for selected features - III

System	Coroutines	Testing	Debugger	Global Variables	Mutable Terms
B-Prolog	✓	✗	trace	✓	✗
Ciao	✓	✓	trace / source	✓	✓
ECLiPSe	✓	✓	trace	✓	✗
GNU Prolog	✗	✗	trace	✓	✓
JIProlog	✗	✗	trace	✗	✗
SICStus	✓	✓	trace / source	✗	✓
SWI	✓	✓	trace / graphical	✓	✓
τProlog	✗	✗	✗	✗	✗
tuProlog	✗	✗	spy	✗	✗
XSB	✓	✗	trace	✗	✗
YAP	✗	✗	trace	✓	✗

Summary

- ISO standard generally supported, however with minor differences.
- A good number of commonly available features:
  - module system
  - constraints
  - multi-threading (interfaces may differ)
  - tabling
  - co-routining

  Module pretty compatible but rest interfaces, domains may differ.
  CHR quite widespread.
- Type and mode annotations used for documentation but not enforced (except Ciao). Approaches:
  - Strong typing (Goedel, Mercury).
  - The Ciao model (now also known as "gradual typing").

Influences on Other Systems

Influence in other languages within LP and its extensions

- Datalog
- λ-Prolog
- Commited-choice languages
- Turbo Prolog
- Mercury
- Goedel
- Curry
- Assumptive LP
- ASP
- s(ASP) and s(CASP)
- CHR, CHRG
- HyProlog
- Co-inductive LP
- Probabilistic LP
- LogTalk
- Picat
- ...

Influence beyond LP

- Theorem proving technology.
- Java (abstract machine, specification, ...).
- Erlang.
- Many embeddings in other languages.
- Many others: C++, many compilers, ...
- Analyzers for other languages.
- ...

Analysis of the current situation and the way forward

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.

- Successful applications, including:
    - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
    - Domain-specific languages.
    - Heterogeneous data integration.
    - Natural language processing.
    - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
    - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
    - Domain-specific languages.
    - Heterogeneous data integration.
    - Natural language processing.
    - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with “declarative” pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
    - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
    - Domain-specific languages.
    - Heterogeneous data integration.
    - Natural language processing.
    - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog strengths

- Clean, simple syntax and semantics.
- Immutable persistent data structures, with "declarative" pointers (logic variables).
- Arbitrary precision arithmetic.
- Safety (garbage collection, no NullPointer exceptions, ...).
- Tail-recursion and last-call optimization.
- Efficient inference, pattern matching, and unification; DCGs.
- Meta-programming, programs as data.
- Constraint solving.
- Independence of the selection rule (coroutines).
- Indexing, efficient tabling.
- Fast development, REPL (Read, Execute, Print, Loop), debugging, ...
- Commercial and open-source systems (some very substantive and mature!).
- Active developer community with constant new implementations, features, etc.
- Sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- Many books, courses, and learning materials.
- Successful applications, including:
  - Program analysis (Abstr. Interp., Set-Based Anal., Datalog, energy, gas, ...).
  - Domain-specific languages.
  - Heterogeneous data integration.
  - Natural language processing.
  - Efficient inference (expert systems, theorem provers), symbolic AI, ...

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!
- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from “traditional” programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Prolog weaknesses → and how to address them

- Learning curve, beginners can easily write programs that loop or consume a huge amount of resources → teach it well, use the right tools! (see later)
- Lack of static typing → but notable exceptions!
- Lack of data hiding → but notable exceptions!
- Lack of object orientation. → but notable exceptions!
- Packages: availability and management → improve compatibility.
- Limited support for embedded or app development → but notable exceptions!

- Syntactically different from "traditional" programming languages, not a mainstream language → offer alternative syntax?
- IDEs and development tools: much progress but still limitations in some areas (e.g., refactoring) → future work?
- Limitations in portability across systems → need to improve.
- UI development (usually conducted in a foreign language via FLI) → exceptions / need to improve?

Summary: much can be taken from other Prolog systems; also work still needed.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an “old” language.
- Wrong image due to “shallow” teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Threats to Prolog's future → and how to address them

- Comparatively small user base.
- Fragmented community with limited interactions.
- Active developer community with constant new implementations, features.
- Further fragmentation of Prolog implementations.
- New programming languages.
- Post-desktop world of JavaScript web-applications.
- The perception that it is an "old" language.
- Wrong image due to "shallow" teaching of the language.

→

- Many weaknesses already addressed by different systems. → continue cooperative/competitive evolution (vs. going for single system).
- But, good forum needed for discussion.
- Also, bring together community across systems.
- Again, improved teaching.

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.

- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation (provably correct refactoring).
    - Parallelism.

  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation (provably correct refactoring?).
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
    - Neuro-Symbolic AI.
    - Explainable AI, verifiable AI.
    - Big Data.
- New features and developments:
    - Probabilistic reasoning.
    - Embedding ASP and SAT or SMT solving, s(CASP) applications.
    - Opportunity still for performance gains (and we have the technology):
        - Full-fledged JIT compiler.
        - Global optimization, partial evaluation ("provably correct refactoring").
        - Parallelism.
    - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
    - Neuro-Symbolic AI.
    - Explainable AI, verifiable AI.
    - Big Data.

- New features and developments:
    - Probabilistic reasoning.
    - Embedding ASP and SAT or SMT solving, s(CASP) applications.
    - Opportunity still for performance gains (and we have the technology):
        - Full-fledged JIT compiler.
        - Global optimization, partial evaluation ("provably correct refactoring").
        - Parallelism.
    - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
    - Neuro-Symbolic AI.
    - Explainable AI, verifiable AI.
    - Big Data.

- New features and developments:
    - Probabilistic reasoning.
    - Embedding ASP and SAT or SMT solving, s(CASP) applications.
    - Opportunity still for performance gains (and we have the technology):
        - Full-fledged JIT compiler.
        - Global optimization, partial evaluation ('provably correct refactoring').
        - Parallelism.
    - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.

  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.
- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.

- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.

  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.

- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.

- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.

  - ...

Opportunities for Prolog

- New application areas, addressing societal challenges:
  - Neuro-Symbolic AI.
  - Explainable AI, verifiable AI.
  - Big Data.

- New features and developments:
  - Probabilistic reasoning.
  - Embedding ASP and SAT or SMT solving, s(CASP) applications.
  - Opportunity still for performance gains (and we have the technology):
    - Full-fledged JIT compiler.
    - Global optimization, partial evaluation ('provably correct refactoring').
    - Parallelism.
  - ...

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
    - ISO, vs. Prolog Commons, vs. future initiatives,
    - Library infrastructure and conditional code,
    - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
-
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
-
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
-
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
-
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Some issues that need joint attention and agreement

- Improve portability of existing features (cf., Prolog systems tables):
  - ISO, vs. Prolog Commons, vs. future initiatives,
  - Library infrastructure and conditional code,
  - Standard test suites beyond ISO.
- Module system (some aspects), interfaces, objects.
- More unified macro system.
- Improved syntactic support for data structures.
- Support for functional programming syntax.
- Types and modes, and other properties.
- Reactivity.
- ... (also, community infrastructure, see at the end).

Dynamic vs. Static — An Almost Religious Argument!

Dynamic languages (Prolog, Lisp/Scheme, Python, Javascript, ...)

- Dynamic checking of types (and many other properties):
  - `..., A is B+C, ...`
    `B` and `C` checked to be `numexpr` by `is/2` at run time.
  - `..., arg(N,T,A), ...`
    `N` checked to be `nat` & $\leq$ `arity(T)` by `arg/3` (array bounds).
- Need to use tags (*boxing* of data) to identify type, var/nonvar, etc.
- Flexibility, compactness, rapid prototyping, scripting, ...

Static languages (ML, Haskell, Mercury, Gödel, ...)

- Compiler checks statically *types*.
- No dynamic checks needed for types.
- Safety guarantees (types), scalability, performance, large systems, ...

- Some languages (e.g., C) are neither (even if still very useful!):
  no checking of, e.g., array bounds at compile time or run time...

Dynamic vs. Static — An Almost Religious Argument!

Dynamic languages (Prolog, Lisp/Scheme, Python, Javascript, ...)

- Dynamic checking of types (and many other properties):
  - `..., A is B+C, ...`
    `B` and `C` checked to be `numexpr` by `is/2` at run time.
  - `..., arg(N,T,A), ...`
    `N` checked to be `nat` & $\leq$ `arity(T)` by `arg/3` (array bounds).
- Need to use tags (*boxing* of data) to identify type, var/nonvar, etc.
- Flexibility, compactness, rapid prototyping, scripting, ...

Static languages (ML, Haskell, Mercury, Gödel, ...)

- Compiler checks statically *types*.
- No dynamic checks needed for types.
- Safety guarantees (types), scalability, performance, large systems, ...

- Some languages (e.g., C) are neither (even if still very useful!):
  no checking of, e.g., array bounds at compile time or run time...

Dynamic vs. Static — An Almost Religious Argument!

Dynamic languages (Prolog, Lisp/Scheme, Python, Javascript, ...)

- Dynamic checking of types (and many other properties):
  - `..., A is B+C, ...`
    `B` and `C` checked to be `numexpr` by `is/2` at run time.
  - `..., arg(N,T,A), ...`
    `N` checked to be `nat` & $\leq$ `arity(T)` by `arg/3` (array bounds).
- Need to use tags (*boxing* of data) to identify type, var/nonvar, etc.
- Flexibility, compactness, rapid prototyping, scripting, ...

Static languages (ML, Haskell, Mercury, Gödel, ...)

- Compiler checks statically *types*.
- No dynamic checks needed for types.
- Safety guarantees (types), scalability, performance, large systems, ...

- Some languages (e.g., C) are neither (even if still very useful!):
  no checking of, e.g., array bounds at compile time or run time...

Solving the Dynamic vs. Static Dilemma

The Ciao Approach:

- Provide the flexibility of dynamic languages, but with
- *Guaranteed safety, reliability, and efficiency.*

- Use of *voluntary assertions* to express desired properties (incl. types).
  - Can be added up front, gradually, or not at all.
- Use of *advanced program analysis* (abstract interpretation) for:
  - Guaranteeing the properties as much as possible at compile-time.
  - Achieving high performance:
    - Eliminating run time checks at compile time.
    - Unboxing.
    - Specialization, slicing, ...
    - Automatic parallelization.

- Integrated Approach to Specification, Verification, Testing, Debugging, and Optimization.

Solving the Dynamic vs. Static Dilemma

The Ciao Approach:

- Provide the flexibility of dynamic languages, but with
- *Guaranteed safety, reliability, and efficiency.*

- Use of *voluntary assertions* to express desired properties (incl. types).
  - Can be added up front, gradually, or not at all.
- Use of *advanced program analysis* (abstract interpretation) for:
  - Guaranteeing the properties as much as possible at compile-time.
  - Achieving high performance:
    - Eliminating run time checks at compile time.
    - Unboxing.
    - Specialization, slicing, ...
    - Automatic parallelization.

- Integrated Approach to Specification, Verification, Testing, Debugging, and Optimization.

Solving the Dynamic vs. Static Dilemma

The Ciao Approach:

- Provide the flexibility of dynamic languages, but with
- *Guaranteed safety, reliability, and efficiency.*

- Use of *voluntary assertions* to express desired properties (incl. types).
  - Can be added up front, gradually, or not at all.
- Use of *advanced program analysis* (abstract interpretation) for:
  - Guaranteeing the properties as much as possible at compile-time.
  - Achieving high performance:
    - Eliminating run time checks at compile time.
    - Unboxing.
    - Specialization, slicing, ...
    - Automatic parallelization.

- Integrated Approach to Specification, Verification, Testing, Debugging, and Optimization.

The Assertion Language

Assertions:

:- pred Pred [:Precond] [=> Postcond] [+ CompProps] .

Example:

```
:- pred quicksort(X,Y) : list(int) * var => sorted(Y) + (is_det,not_fails).
:- pred quicksort(X,Y) : var * list(int) => ground(X) + non_det.
```

- Optional, can be added at any time. Provide partial specification.
- Describe calls, success, and computational behavior/invariants.
- Each `pred` typically describes a “mode” of use; the set *covers all valid calls*.
- System makes it worthwhile for the programmer to use them: e.g., autodoc.

Inst vs. Compat:

- The **:** and => fileds describe *instantiation states* by default.
- Specifying “compatibility:”

```
:- pred quicksort/2 :: list(int) * list(int).
```

The Assertion Language (Contd.)

Properties:

```
:- regtype color := green | blue | red.
:- regtype list := [] | [_|list].
:- regtype list(X) := [] | [ X|list].          ≡   list(_,[]). list(X,[H|T]) :- X(H), list(X,T).
:- prop sorted := [] | [ _ ] | [X,Y|Z] :- X > Y, sorted([Y|Z]).
```

- Arbitrary predicates (but conditions on them: termination, steadfastness, ...)
- Many predefined in libs, some of them "native" to an analyzer.
  Can also be user-defined.
- Should be visible/imported and "runnable:" used also as run-time tests!
- *Types/shapes* are a special case of property (e.g., `regtype`s).
- But also, e.g., data sizes, instantiation states, aliasing, termination, determinacy, non-failure, time, memory, ...

The Assertion Language (Contd.)

Modes (essentially “property macros”):

```
:- pred qs(+,-).            ≡   :- pred qs(X,Y) :  (nonvar(X), var(Y)).
:- pred qs(?list,?list).    ≡   :- pred qs(X,Y) :: (list(X), list(Y)).
:- pred qs(+list,-list).    ≡   :- pred qs(X,Y) :  (list(X), var(Y)) => list(Y).
```

In fact, they are defined as macros:

```
:- modedef +(A) : nonvar(A).        :- modedef +(A,X) : X(A).
:- modedef -(A) : var(A).           :- modedef -(A,X) : var(A) => X(A).
```

Can include comments:

```
:- pred qs(+list,-list)   #   "Sorts."
:- pred qs(-list,+list)   #   "Generates permutations."
```

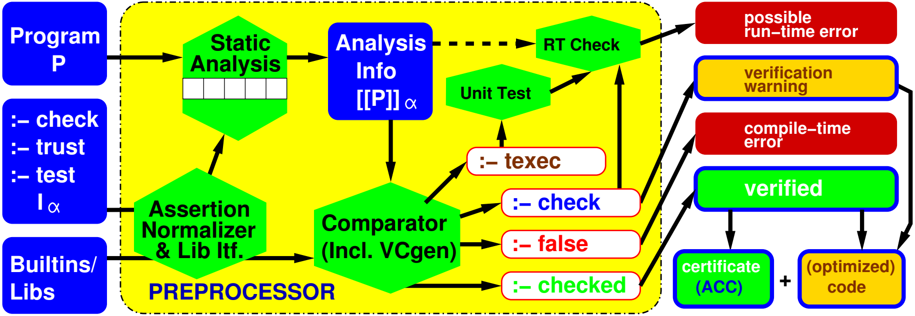
Program-point Assertions:

- Inlined with code: `..., check(( int(X), X>0 )), ....`

Assertion Status (so far “to be checked” – check status – default):

- Other: **trust** (guide analyzer), **true**/**false** (analysis output), **test**, etc.

The Ciao Integrated Approach to Specification, Debugging, Verification, Testing, and Optimization (Mostly Mid 90's!)



Discussion: Comparison with *Classical* Types

“Traditional” Types	**Ciao Assertion-based Model**
“Properties” limited by decidability	Much more general property language
May need to limit prog. lang.	No need to limit prog. lang.
“Untypable” programs rejected	Run-time checks introduced
(Almost) Decidable	Decidable + Undecidable (approximated)
Expressed in a different language	Expressed in the source language
Types must be defined	Types can be defined or inferred
Assertions are only of type “check”	“check”, “trust”, ...
Type signatures & assertions different	Type signatures *are* assertions

- But quite popular now: gradual typing, Racket, liquid Haskell, etc.
- Some key issues:
  Safe / Sound approximation *Suitable assertion language*
  Abstract Interpretation *Powerful abstract domains*
- Works best when properties and assertions can be expressed in the source language (i.e., source lang. supports *predicates*, *constraints*).

Discussion: Comparison with *Classical* Types

“Traditional” Types	**Ciao Assertion-based Model**
“Properties” limited by decidability	Much more general property language
May need to limit prog. lang.	No need to limit prog. lang.
“Untypable” programs rejected	Run-time checks introduced
(Almost) Decidable	Decidable + Undecidable (approximated)
Expressed in a different language	Expressed in the source language
Types must be defined	Types can be defined or inferred
Assertions are only of type “check”	“check”, “trust”, ...
Type signatures & assertions different	Type signatures *are* assertions

- But quite popular now: gradual typing, Racket, liquid Haskell, etc.
- Some key issues:
  Safe / Sound approximation *Suitable assertion language*
  Abstract Interpretation *Powerful abstract domains*
- Works best when properties and assertions can be expressed in the source language (i.e., source lang. supports *predicates*, *constraints*).

Discussion: Comparison with *Classical* Types

“Traditional” Types	**Ciao Assertion-based Model**
“Properties” limited by decidability	Much more general property language
May need to limit prog. lang.	No need to limit prog. lang.
“Untypable” programs rejected	Run-time checks introduced
(Almost) Decidable	Decidable + Undecidable (approximated)
Expressed in a different language	Expressed in the source language
Types must be defined	Types can be defined or inferred
Assertions are only of type “check”	“check”, “trust”, ...
Type signatures & assertions different	Type signatures *are* assertions

- But quite popular now: gradual typing, Racket, liquid Haskell, etc.
- Some key issues:
  Safe / Sound approximation *Suitable assertion language*
  Abstract Interpretation *Powerful abstract domains*
- Works best when properties and assertions can be expressed in the source language (i.e., source lang. supports *predicates*, *constraints*).

Teaching (and preaching) Prolog

On teaching Prolog

- Prolog / LP / CLP *must* be taught in CS programs,
  - A CS graduate is simply not complete without knowledge of Prolog.

  (and maybe also in other majors and maybe in schools –cf. Prolog Year?)
- But it has to be done right!
  - The standard 'programming paradigms' approach is counter-productive.
  - Simply cannot be done in a couple of weeks emulating Prolog in Scheme.
    - What to do if that is the only slot available?
- On the way *dispel unfounded myths* about the language, and show how many of the shortcomings of early Prologs have been *addressed over the years*.

On teaching Prolog

- Prolog / LP / CLP *must* be taught in CS programs,
  - A CS graduate is simply not complete without knowledge of Prolog.

  (and maybe also in other majors and maybe in schools –cf. Prolog Year?)

- But it has to be done right!
  - The standard 'programming paradigms' approach is counter-productive.
  - Simply cannot be done in a couple of weeks emulating Prolog in Scheme.
    - What to do if that is the only slot available?

- On the way *dispel unfounded myths* about the language, and show how many of the shortcomings of early Prologs have been *addressed over the years*.

On teaching Prolog

- Prolog / LP / CLP *must* be taught in CS programs,
  - A CS graduate is simply not complete without knowledge of Prolog.

  (and maybe also in other majors and maybe in schools –cf. Prolog Year?)
- But it has to be done right!
  - The standard 'programming paradigms' approach is counter-productive.
  - Simply cannot be done in a couple of weeks emulating Prolog in Scheme.
    - What to do if that is the only slot available?
- On the way *dispel unfounded myths* about the language, and show how many of the shortcomings of early Prologs have been *addressed over the years*.

On teaching Prolog

- "Prolog gets into infinite loops."

  This is true –in fact, of any programming language or proof system. However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other languages) can solve that (but tabling helps).
  - Discuss advantages and disadvantages (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

On teaching Prolog

- “Prolog gets into infinite loops.”

  This is true –in fact, of any programming language or proof system. However, it is likely to discourage beginners if not explained well:

    - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
    - Start by running all predicates, e.g., breadth-first – everything works!
    - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
    - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other languages) can solve that (but tabling helps).
    - Discuss advantages and disadvantages (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

On teaching Prolog

- “Prolog gets into infinite loops.”

  This is true –in fact, of any programming language or proof system.
  However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other languages) can solve that (but tabling helps).
  - Discuss advantages and disadvantages (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

On teaching Prolog

- "Prolog gets into infinite loops."

  This is true –in fact, of any programming language or proof system. However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other languages) can solve that (but tabling helps).
  - Discuss advantages and disadvantages (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

On teaching Prolog

- “Prolog gets into infinite loops.”

  This is true –in fact, of any programming language or proof system. However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other languages) can solve that (but tabling helps).
  - Discuss advantages and disadvantages (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

On teaching Prolog

- "Prolog gets into infinite loops."

  This is true –in fact, of any programming language or proof system. However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other languages) can solve that (but tabling helps).
  - Discuss advantages and disadvantages (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

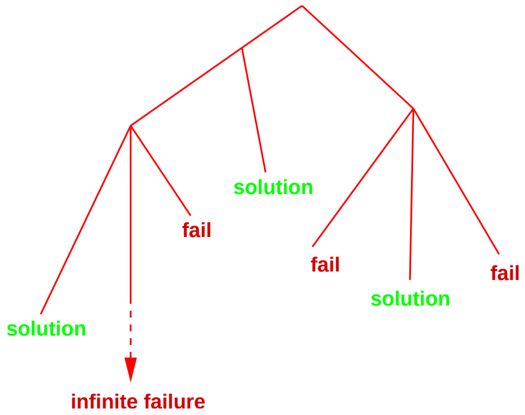
On teaching Prolog

- "Prolog gets into infinite loops."

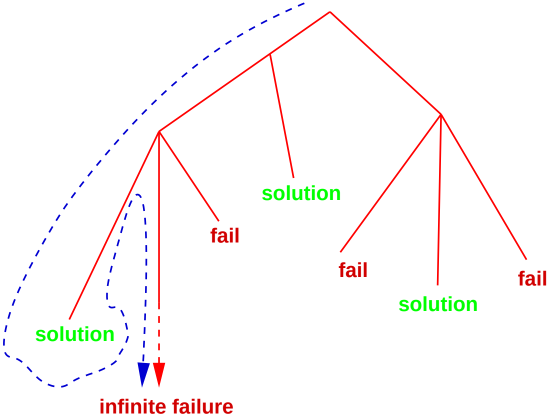
  This is true –in fact, of any programming language or proof system.
  However, it is likely to discourage beginners if not explained well:

  - Use a system that can *alternatively and selectively* run in breadth-first, iterative deepening, tabling, etc.
  - Start by running all predicates, e.g., breadth-first – everything works!
  - Then, explain the shape of the tree (solutions at finite depth, possible infinite failures, etc.), and thus why breadth-first works, and why depth-first sometimes may not.
  - Do relate it to the *halting problem*: no-one (Prolog, logic, nor other languages) can solve that (but tabling helps).
  - Discuss advantages and disadvantages (time, memory). Motivate the choices made for Prolog benchmarking actual executions.

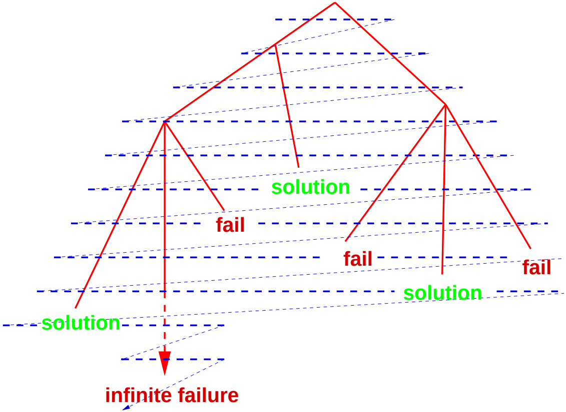
Characterization of the search tree



Depth-First Search



Breadth-First Search



On teaching Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- “Arithmetic is not reversible.”
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- “There is no occur check.”
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- “Prolog is not pure (cut, assert, etc.)”
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- “Arithmetic is not reversible.”
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- “There is no occur check.”
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- “Prolog is not pure (cut, assert, etc.)”
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- “Arithmetic is not reversible.”
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- “There is no occur check.”
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- “Prolog is not pure (cut, assert, etc.)”
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- “Arithmetic is not reversible.”
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- “There is no occur check.”
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- “Prolog is not pure (cut, assert, etc.)”
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- "Arithmetic is not reversible."
    - Start with Peano arithmetic: beautiful but slow.
    - Then justify Prolog arithmetic for efficiency.
    - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
    - Explain why, and that there is a built-in for it.
    - Have a package (expansion) that calls it by default for all unifications.
    - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
    - Have a pure mode in the implementation so that impure built-ins simply are not present.
    - Develop pure libraries (including monad-style).
    - Develop purer built-ins.

and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- “Arithmetic is not reversible.”
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- “There is no occur check.”
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- “Prolog is not pure (cut, assert, etc.)”
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- “Arithmetic is not reversible.”
    - Start with Peano arithmetic: beautiful but slow.
    - Then justify Prolog arithmetic for efficiency.
    - Then show (arithmetic) constraint domains: beautiful and efficient!
- “There is no occur check.”
    - Explain why, and that there is a built-in for it.
    - Have a package (expansion) that calls it by default for all unifications.
    - Explain the existence of infinite tree unification (as a constraint domain).
- “Prolog is not pure (cut, assert, etc.)”
    - Have a pure mode in the implementation so that impure built-ins simply are not present.
    - Develop pure libraries (including monad-style).
    - Develop purer built-ins.

and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- “Arithmetic is not reversible.”
    - Start with Peano arithmetic: beautiful but slow.
    - Then justify Prolog arithmetic for efficiency.
    - Then show (arithmetic) constraint domains: beautiful and efficient!
- “There is no occur check.”
    - Explain why, and that there is a built-in for it.
    - Have a package (expansion) that calls it by default for all unifications.
    - Explain the existence of infinite tree unification (as a constraint domain).
- “Prolog is not pure (cut, assert, etc.)”
    - Have a pure mode in the implementation so that impure built-ins simply are not present.
    - Develop pure libraries (including monad-style).
    - Develop purer built-ins.

  and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- “Arithmetic is not reversible.”
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- “There is no occur check.”
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- “Prolog is not pure (cut, assert, etc.)”
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- "Arithmetic is not reversible."
  - Start with Peano arithmetic: beautiful but slow.
  - Then justify Prolog arithmetic for efficiency.
  - Then show (arithmetic) constraint domains: beautiful and efficient!
- "There is no occur check."
  - Explain why, and that there is a built-in for it.
  - Have a package (expansion) that calls it by default for all unifications.
  - Explain the existence of infinite tree unification (as a constraint domain).
- "Prolog is not pure (cut, assert, etc.)"
  - Have a pure mode in the implementation so that impure built-ins simply are not present.
  - Develop pure libraries (including monad-style).
  - Develop purer built-ins.

and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

- “Arithmetic is not reversible.”
    - Start with Peano arithmetic: beautiful but slow.
    - Then justify Prolog arithmetic for efficiency.
    - Then show (arithmetic) constraint domains: beautiful and efficient!
- “There is no occur check.”
    - Explain why, and that there is a built-in for it.
    - Have a package (expansion) that calls it by default for all unifications.
    - Explain the existence of infinite tree unification (as a constraint domain).
- “Prolog is not pure (cut, assert, etc.)”
    - Have a pure mode in the implementation so that impure built-ins simply are not present.
    - Develop pure libraries (including monad-style).
    - Develop purer built-ins.

and accept that impurity is necessary sometimes, but keep it encapsulated when possible.

On teaching Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
    - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
      (This idea useful for analysis of other languages!)
    - Show that it is completely normal if used in one direction and there is only one definition per procedure.
    - But it can also have several definitions, search, run backwards, etc.
    - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
    - The TIOBE index disagrees...
    - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- “Prolog is a strange language.”
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- “Prolog has no applications / interest / nobody uses it.”
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- “The Fifth Generation failed!” Not true...
  and it did not use Prolog or “real LP” anyway!
  They used in fact “something like Erlang”
  (probably why it was not as successful as it could have been.)

On teaching Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- “Prolog is a strange language.”
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- “Prolog has no applications / interest / nobody uses it.”
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- “The Fifth Generation failed!” Not true...

and it did not use Prolog or “real LP” anyway!

They used in fact “something like Erlang”

(probably why it was not as successful as it could have been.)

On teaching Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- “Prolog is a strange language.”
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- “Prolog has no applications / interest / nobody uses it.”
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- “The Fifth Generation failed!” Not true...
  and it did not use Prolog or “real LP” anyway!
  They used in fact “something like Erlang”
  (probably why it was not as successful as it could have been.)

On teaching Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- “Prolog is a strange language.”
    - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
      (This idea useful for analysis of other languages!)
    - Show that it is completely normal if used in one direction and there is only one definition per procedure.
    - But it can also have several definitions, search, run backwards, etc.
    - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- “Prolog has no applications / interest / nobody uses it.”
    - The TIOBE index disagrees...
    - Show some good examples of applications (cf. Prolog Year).
- “The Fifth Generation failed!” Not true...
  and it did not use Prolog or “real LP” anyway!
  They used in fact “something like Erlang”
  (probably why it was not as successful as it could have been.)

On teaching Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...

  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching Prolog

The following views are specially relevant to teaching Prolog (and LP) to (CS) college students: they have already been exposed to other languages (imperative/OO, sometimes functional) and probably have some notions of PL implementation.

- "Prolog is a strange language."
  - Show that Prolog *subsumes* functional and imperative programming (after SSA). It is simply *that and more*.
    (This idea useful for analysis of other languages!)
  - Show that it is completely normal if used in one direction and there is only one definition per procedure.
  - But it can also have several definitions, search, run backwards, etc.
  - In addition to a stack of forward continuations, as every language, to know where go when a procedure returns (succeeds), it also has a stack of *backwards continuations* to go if there is a failure (previous *choice point*).
- "Prolog has no applications / interest / nobody uses it."
  - The TIOBE index disagrees...
  - Show some good examples of applications (cf. Prolog Year).
- "The Fifth Generation failed!" Not true...
  and it did not use Prolog or "real LP" anyway!
  They used in fact "something like Erlang"
  (probably why it was not as successful as it could have been.)

On teaching Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching Prolog

- Do show the beauty:
  - Explain “Green’s dream,” discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don’t matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving “the data structures class.”
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or “run backwards” to generate the “inhabitants”; property-based testing for free!

On teaching Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching Prolog

- Do show the beauty:
  - Explain "Green's dream," discuss for what logics we have effective deduction procedures, justify the choice of FO and semi-decidability, SLD-resolution → classical LP (Kowalski/Colmerauer).
  - Show how logic programs are both logical theories (with declarative meaning) and procedural programs that can be debugged, followed step by step, etc.
    - An operational (in addition to declarative) semantics is a requirement in the language (vs., e.g., Goedel) and we do need to teach it.
    - Otherwise not a programming language, just specification/KR – Prolog is both.
    - How otherwise to reason about complexity, memory consumption, etc.? To say that these things don't matter does not make sense in PL.
  - Show with examples (and benchmarking them) how you can go from executable specifications to efficient algorithms gradually, and as needed.
  - Show how unification is also a device for *constructing and matching complex data structures with (declarative) pointers*. Show it in the top level, giving "the data structures class."
  - Do use types (and properties in general): define them as predicates, show them used to check if something is in the type (dynamic checking), or "run backwards" to generate the "inhabitants"; property-based testing for free!

On teaching Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
- Ideally system allows covering: pure LP (with several search rules), ISO-Prolog, constraints, HO, etc.

On teaching Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
- Ideally system allows covering: pure LP (with several search rules), ISO-Prolog, constraints, HO, etc.

On teaching Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
- Ideally system allows covering: pure LP (with several search rules), ISO-Prolog, constraints, HO, etc.

On teaching Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
- Ideally system allows covering: pure LP (with several search rules), ISO-Prolog, constraints, HO, etc.

On teaching Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / “real” use.
    Show serious, competitive language.
  - Playgrounds and notebooks (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
- Ideally system allows covering: pure LP (with several search rules), ISO-Prolog, constraints, HO, etc.

On teaching Prolog

- System types:
  - Classical installation.
    Most appropriate for more advanced students / "real" use.
    Show serious, competitive language.
  - Playgrounds and notebooks (e.g., SWISH, Ciao Playgrounds/active manuals, τ-Prolog).
    - Server-based.
    - Browser-based.

    Can be attractive for beginners, young students.
- Ideally system allows covering: pure LP (with several search rules), ISO-Prolog, constraints, HO, etc.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.

Final thoughts

- The classical characteristics of Prolog are still unique and demanded.
- It is still one of the most interesting computing paradigms.
- Plus, it is also not 'your grandfather's Prolog' any more.
- Many (most?) of the initial shortcomings of the language have been addressed, even if by different systems.
- More relevant than ever at a time in need for explainable AI.

- Regarding system coordination: despite the intense evolution, differences between systems are not fundamental. To progress:
  - Create a forum (e.g., a web platform) to discuss proposals and gather pointers to solutions, in order to reach consensus on the most important extensions of current implementations.
  - Also, a structured workflow for tracking proposals.
  - Take advantage of / build on existing mechanisms such as the ISO standard or (an updated version of) the Prolog Commons.
  - Involve implementors and users.
  - Under the wings of ALP.
  - Some parts of it can result from the Year of Prolog efforts.