Syntax Analysis. Data Types and Expressions.

Outline

- ◇ Syntactic Analysis
- ◇ Datatypes
- ◇ Expressions

Syntactic Analysis

- Based on unambiguous grammar
- Check for syntactic correctness
- Construct *abstract syntax tree*
- Linear complexity is desired

Reasoning Rules

- Formal tool to specify reasoning processes in many areas: logic, programming languages, semantics, verification, etc.

- Format:

$$\frac{\mathit{context}_1 \vdash \text{judgement}_1 \ldots \mathit{context}_n \vdash \text{judgement}_n}{\mathit{context} \vdash \text{judgement}} \quad \textit{side conditions}$$

- Prolog rules have a similar reading, and thus can serve as straightforward implementation.

Reasoning Rules

$\boxed{\text{Premises}}$

- Formal tool to specify reasoning processes in many areas: logic, programming languages, semantics, verification, etc.

- Format:

$$\frac{\mathit{context}_1 \vdash \text{judgement}_1 \ \ldots \ \mathit{context}_n \vdash \text{judgement}_n}{\mathit{context} \vdash \text{judgement}} \quad \textit{side conditions}$$

- Prolog rules have a similar reading, and thus can serve as straightforward implementation.

$\boxed{\text{Conclusion}}$

Reasoning Rules

- Formal tool to specify reasoning processes in many areas: logic, programming languages, semantics, verification, etc.

- Format:

$$\frac{\textit{context}_1 \vdash \text{judgement}_1 \ldots \textit{context}_n \vdash \text{judgement}_n}{\textit{context} \vdash \text{judgement}} \quad \textit{side conditions}$$

- Prolog rules have a similar reading, and thus can serve as straightforward implementation.

If judgement 1 is true in context 1 and... judgement n is true in context n, then judgement is true in context.

Reasoning Rules

Explain notations that appear in the reasoning process.

- Formal tool to specify reasoning processes in many areas: logic, programming languages, semantics, verification, etc.
- Format:

$$\frac{\mathit{context}_1 \vdash \text{judgement}_1 \ldots \mathit{context}_n \vdash \text{judgement}_n}{\mathit{context} \vdash \text{judgement}} \quad \textit{side conditions}$$

- Prolog rules have a similar reading, and thus can serve as straightforward implementation.

If judgement 1 is true in context 1 and... judgement n is true in context n, then judgement is true in context.

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \hat{\ }\, s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \hat{}\, s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

If string s_1 is generated by nonterminal <subexpr>

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \qquad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \qquad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \qquad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \qquad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \qquad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \qquad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \hat{}\ s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

If string s_1 is generated by nonterminal <subexpr>

And string s_2 is generated by nonterminal <term>

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>} \vdash s_1 \quad \texttt{<term>} \vdash s_2}{\texttt{<expr>} \vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>} \vdash s_1 \quad \texttt{<term>} \vdash s_2}{\texttt{<subexpr>} \vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>} \vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>} \vdash s_1 \quad \texttt{<factor>} \vdash s_2}{\texttt{<term>} \vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>} \vdash s_1 \quad \texttt{<factor>} \vdash s_2}{\texttt{<subterm>} \vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>} \vdash \texttt{<>}}$$

$$\frac{\texttt{<base>} \vdash s_1 \quad \texttt{<restexp>} \vdash s_2}{\texttt{<factor>} \vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>} \vdash s_1 \quad \texttt{<restexp>} \vdash s_2}{\texttt{<subterm>} \vdash \hat{\ }\, s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>} \vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>} \vdash s_1}{\texttt{<base>} \vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>} \vdash s} \quad s \in \{a, \ldots, z\}$$

If string s_1 is generated by nonterminal `<subexpr>`

And string s_2 is generated by nonterminal `<term>`

Then string s is generated by nonterminal `<expr>`

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \qquad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \qquad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \qquad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \qquad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \qquad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \qquad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash\ \hat{}\ s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

If string s_1 is generated by nonterminal **<subexpr>**

And string s_2 is generated by nonterminal **<term>**

Then string s is generated by nonterminal **<expr>**

Where s is s_1 concatenated with s_2.

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \hat{\ }\, s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash \texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{+,-\}$$

$$\frac{}{\texttt{<subexpr>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \quad s = s_1 s_2 s_3,\ s_3 \in \{*,/\}$$

$$\frac{}{\texttt{<subterm>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \quad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \hat{}\ s} \quad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \quad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \quad s \in \{a, \ldots, z\}$$

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash \hat{}\ s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

```
<expr>      ::= <subexpr> <term>
<subexpr>   ::= <subexpr> <term> ['+'|'-']
              | <>
<term>      ::= <subterm> <factor>
<subterm>   ::= <subterm> <factor> ['*'|'/']
              | <>
<factor>    ::= <base> <restexp>
<restexp>   ::= '^' <base> <restexp>
              | <>
<base>      ::= '(' <expr> ')'
              | a | b | c | d
```

Syntactic Analysis as Reasoning Rules

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+, -\}$$

$$\frac{}{\texttt{<subexpr>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<term>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<subterm>}\vdash s_1 \quad \texttt{<factor>}\vdash s_2}{\texttt{<subterm>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{*, /\}$$

$$\frac{}{\texttt{<subterm>}\vdash\texttt{<>}}$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<factor>}\vdash s} \qquad s = s_1 s_2$$

$$\frac{\texttt{<base>}\vdash s_1 \quad \texttt{<restexp>}\vdash s_2}{\texttt{<subterm>}\vdash\ \hat{}\ s} \qquad s = s_1 s_2$$

$$\frac{}{\texttt{<restexp>}\vdash\texttt{<>}}$$

etc...

$$\frac{\texttt{<expr>}\vdash s_1}{\texttt{<base>}\vdash s} \qquad s = (s_1)$$

$$\frac{}{\texttt{<base>}\vdash s} \qquad s \in \{a, \ldots, z\}$$

```
<expr>     ::= <subexpr> <term>
<subexpr>  ::= <subexpr> <term> ['+'|'-']
             | <>
<term>     ::= <subterm> <factor>
<subterm>  ::= <subterm> <factor> ['*'|'/']
             | <>
<factor>   ::= <base> <restexp>
<restexp>  ::= '^' <base> <restexp>
             | <>
<base>     ::= '(' <expr> ')'
             | a | b | c | d
```

Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

Prolog double quoted term: list of ASCII codes:

`"abc" = [97,98,99]`

Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

List of lists

Prolog double quoted term:
list of ASCII codes:

`"abc" = [97,98,99]`

Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<expr>}\vdash s} \qquad s = s_1 s_2$$

Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

$$\frac{\texttt{<subexpr>}\vdash s_1 \quad \texttt{<term>}\vdash s_2}{\texttt{<subexpr>}\vdash s} \qquad s = s_1 s_2 s_3,\ s_3 \in \{+,-\}$$

Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

$\overline{\texttt{<subexpr>} \vdash \texttt{<>}}$

Prolog Code — Attempt 1

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

Empty string is prefix of empty string.

Left recursion

Runs into infinite loop !!!

Prolog Code — Attempt 1

Add heuristics!

Non-empty
Balanced brackets
No + or - outside
bracktes

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

Prolog Code — Attempt 1

Add heuristics!

```
expr(S) :-
    append(S1,S2,S), subexpr(S1), term(S2).

subexpr("").
subexpr(S) :-
    append([S1,S2,O],S), subexpr(S1),term(S2),member(O,["+","-"]).

term(S) :-
    append(S1,S2,S), subterm(S1), factor(S2).

subterm("").
subterm(S):-
    append([S1,S2,O],S), subterm(S1),factor(S2),member(O,["*","/"]).

factor(S) :-
    append(S1,S2,S), base(S1), restexp(S2).

restexp("").
restexp(S) :-
    append(["^",S1,S2],S), base(S1), restexp(S2).

base(S) :-
    append(["(",S1,")"],S), expr(S1).
base([S]) :-
    97 =< S, S =< 122.
```

Non-empty
Balanced brackets
No + or - outside bracktes

Non-empty
Balanced brackets
No * or / outside brackets

Non-empty
Balanced brackets
No ^ outside brackets

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).

balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).
```

```
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).
```

S2 is non-empty

```
subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).
```

```
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

S is concatenation of S1 and S2

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).

balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Prolog Code — Attempt 2

Characters outside brackets

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

No + or - outside brackets in S2

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).

balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Watch demo on `findall` and `append/2`

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).
```

S2 should contain balanced brackets only

```
subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).
```

```
balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
            (   O \= [] -> member(O,OL) ; true ).

balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

S is S1 concatenated with S2 and O, where O is either + or -

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[0],[S1,S2,[0]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[0],[S1,S2,[0]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
        (   O \= [] -> member(O,OL) ; true ).

balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Apply heuristics in all places where it is useful!

Prolog Code — Attempt 2

```
expr(S) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1), term(S2).

subexpr("") :- !.
subexpr(S) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    !,subexpr(S1),term(S2).

term(S) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1), factor(S2).

subterm("") :- !.
subterm(S):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1),factor(S2).

factor(S) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1), restexp(S2).

restexp("") :- !.
restexp(S) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1), restexp(S2).

base(S) :- append(["(",S1,")"],S), !, expr(S1).
base([S]) :- 97 =< S, S =< 122.
```

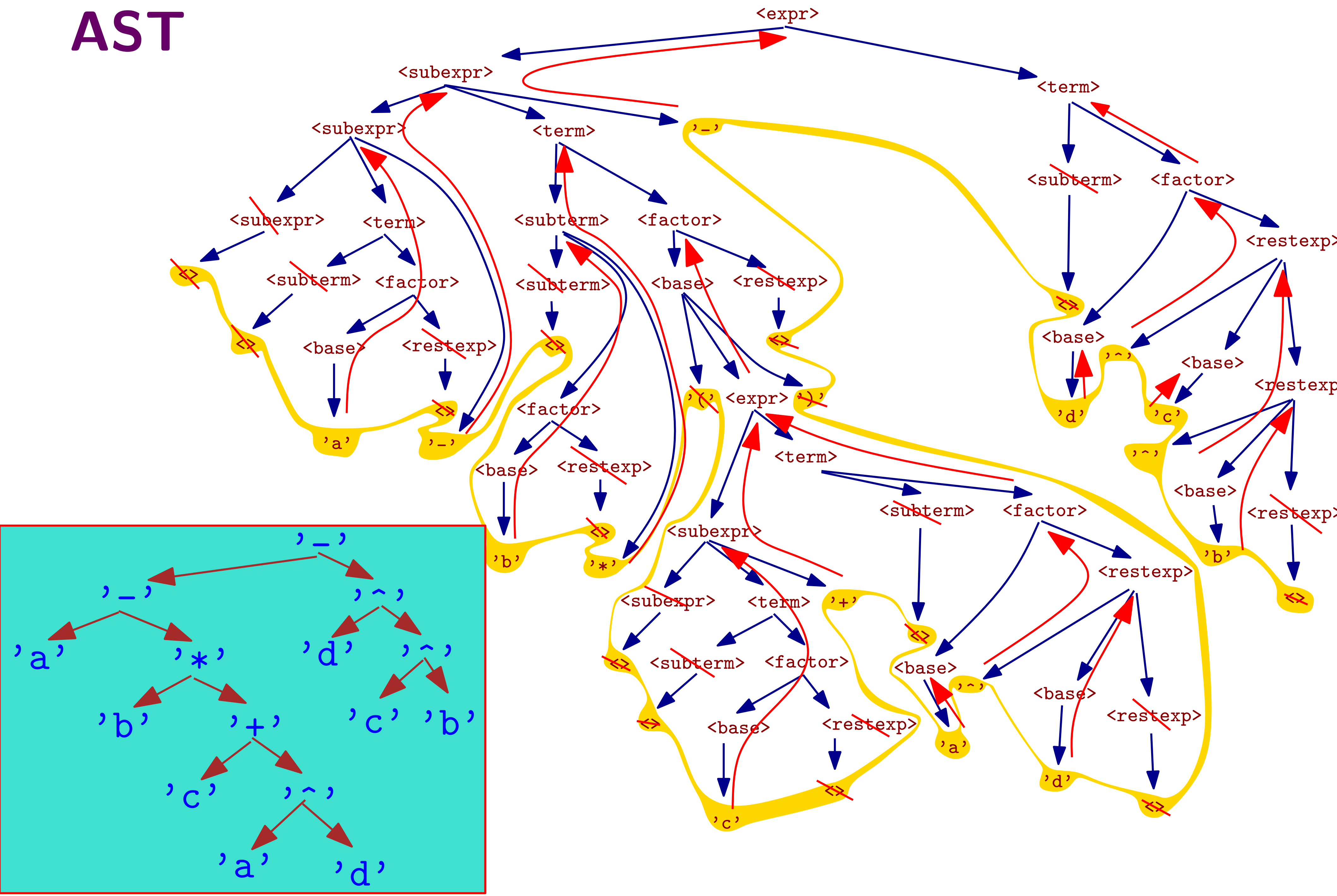
```
constrain(S,S1,O,L,OL) :-
    S1 = [_|_], append(L,S), balanced(S1,R1),
    findall(X,(member([X],OL),member(X,R1)),[]),
         (   O \= [] -> member(O,OL) ; true ).

balanced("","") :- !.
balanced(S,"") :-
    append(["(",S1,")"],S),balanced(S1,_),!.
balanced(S,R) :-
    append([X],S1,S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append([X],R1,R).
balanced(S,R) :-
    append(S1,[X],S), \+ member([X],["(",")"]),!,
    balanced(S1,R1), append(R1,[X],R).
balanced(S,R) :-
    append(["(",S1,")",S2,"(",S3,")"],S),
    balanced(S1,_),balanced(S2,R),balanced(S3,_).
```

Query:

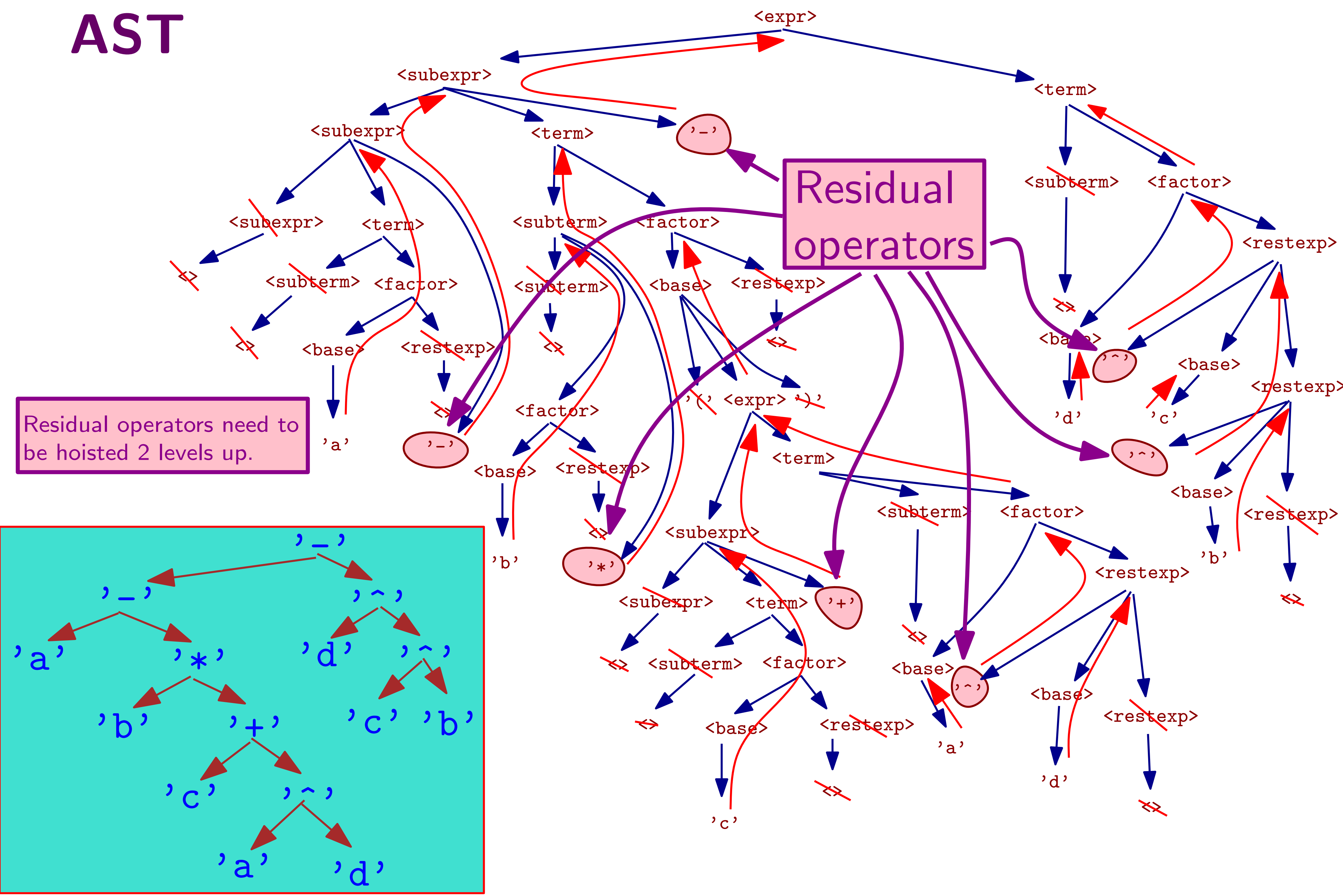
```
1 ?- S="(((a+b)*c/d^e^f-g)^(a*b)+c)*(a+b)", expr(S).
S = [40, 40, 40, 97, 43, 98, 41, 42, 99|...].
```

AST



a - b * (c + a ^ d) - d ^ c ^ b

AST



a - b * (c + a ^ d) - d ^ c ^ b

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Piggyback on the syntax analyzer

Building an $\boxed{\text{AST}}$

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Second argument is where the AST is output

Piggyback on the syntax analyzer

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Residual operator
Third argument of some nonterminals

Piggyback on the syntax analyzer

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Convert ASCII code S into atom A.

Piggyback on the syntax analyzer

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).
```

```
build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Piggyback on the syntax analyzer

Tree building:

- ◇ If the residual operator is `nil`, just pass the current tree up.
- ◇ If the residual operator is not nil, then `L` contains the tree components, which must be assembled into a term.

Building an AST

```
expr(S,T) :-
    constrain(S,S2,[],[S1,S2],["+","-"]),
    !,subexpr(S1,T1,O1), term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subexpr("",nil,nil) :- !.
subexpr(S,T,Op) :-
    constrain(S,S2,[O],[S1,S2,[O]],["+","-"]),
    char_code(Op,O),
    !,subexpr(S1,T1,O1),term(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

term(S,T) :-
    constrain(S,S2,[],[S1,S2],["*","/"]),
    !,subterm(S1,T1,O1), factor(S2,T2),
    build(T,T1,T2,[O1,T1,T2]).

subterm("",nil,nil) :- !.
subterm(S,T,Op):-
    constrain(S,S2,[O],[S1,S2,[O]],["*","/"]),
    !,subterm(S1,T1,O1),factor(S2,T2), char_code(Op,O),
    build(T,T1,T2,[O1,T1,T2]).

factor(S,T) :-
    constrain(S,S1,[],[S1,S2],["^"]),
    !,base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).
```

```
restexp("",nil,nil) :- !.
restexp(S,T,^) :-
    constrain(S,S1,"^",["^",S1,S2],["^"]),
    !, base(S1,T1), restexp(S2,T2,O2),
    build(T,T2,T1,[O2,T1,T2]).

base(S,T) :- append(["(",S1,")"],S), !, expr(S1,T).
base([S],A) :- 97 =< S, S =< 122, char_code(A,S).

build(T,nil,T,_) :- !.
build(T,_,_,L) :- T =.. L .
```

Query:

```
1 ?- S="(((a+b)*c/d^e^f-g)^(a*b)+c)*(a+b)",
      expr(S,T), T =.. L, S =.. X.
S = [40, 40, 40, 97, 43, 98, 41, 42, 99|...],
T = (((a+b)*c/d^e^f-g)^ (a*b)+c)* (a+b),
L = [*, ((a+b)*c/d^e^f-g)^ (a*b)+c, a+b],
X = ['.', 40, [40, 40, 97, 43, 98, 41|...]].
```

What Have We Learned?

- ◇ Reasoning rules are a general formalism for specifying computational mechanisms.
- ◇ Syntax analysis can be specified as reasoning rules.
- ◇ Prolog rules can easily implement reasoning rules.
- ◇ Heuristics need to be employed to make the rules really ”computational”
- ◇ A syntax analyzer can be easily augmented to produce an AST.

Datatypes

- Means of providing interpretation to the "bits".

- Two classes:
  - *Basic:* supported in hardware
  - *Aggregate:* hierarchical way of combining basic types

- In *strongly typed languages* (Ocaml, Haskell): means of detecting incorrect usage of functions

Basic Types

- Most languages have these types. We do a case study on C.
- `int` : 32 bits, signed (2's complement)
- `unsigned int` : 32 bits, unsigned
- `long long int` : 64 bits, signed
- `long long unsigned int` : 64 bits, unsigned
- `short int` : 16 bits, signed
- `unsigned short int` : 16 bits, unsigned
- `char` : 8 bits, signed
- `unsigned char` : 8 bits, unsigned
- `unsigned char` : 8 bits, unsigned
- `float` : 32 bit floating point reals
- `double` : 64 bit floating point reals
- `long double` : 80 bit floating point reals
- Operations on these types will be translated directly into the corresponding machine code instructions: *execution is very efficient*

Datatype Conversions

- Usually called *casts*. Two main types:
  - A notion of *value* can be preserved
    - integral $\longrightarrow$ real : preserve the value
    - real $\longrightarrow$ integral : truncates away the fractional part
    - small size integral $\longrightarrow$ large size integral : *sign extended*
      - Conversion may take the `floor` or the `ceiling`, depending on the language.
  - Bits can be copied over.
    - Conversion between pointer types
    - If sizes are not the same, the least significant bytes are usually preserved
- Implicit casts:
  - Compiler tries to perform conversions to match operators and declarations of functions.
  - When the result of an expression does not fit into 32 bits, it is cast to its address.
  - Exception: structures
  - Specific to C, most other languages do not perform this cast

Datatype Conversions

- Usually called *casts*. Two main types:
  - A notion of *value* can be preserved
    - integral $\longrightarrow$ real : preserve the value
    - real $\longrightarrow$ integral : truncates away the fractional part
    - small size integral $\longrightarrow$ large size integral : *sign extended*
      - Conversion may take the `floor` or the `ceiling`, depending on the language.

```
int a; float b;
...
x = a + b ;
```

`a` is converted to float, value is preserved

  - Bits can be copied over.
    - Conversion between pointer types
    - If sizes are not the same, the least significant bytes are usually preserved

- Implicit casts:
  - Compiler tries to perform conversions to match operators and declarations of functions.
  - When the result of an expression does not fit into 32 bits, it is cast to its address.
  - Exception: structures
  - Specific to C, most other languages do not perform this cast

Datatype Conversions

◇ Usually called *casts*. Two main types:
  – A notion of *value* can be preserved
    ∗ integral → real : preserve the value
    ∗ real → integral : truncates away the fractional part
    ∗ small size integral → large size integral : *sign extended*
      · Conversion may take the `floor` or the `ceiling`, depending on the language.

```
char a; int b;
...
x = a + b ;
```

`a` is sign-extended to int, value is preserved

  – Bits can be copied over.
    ∗ Conversion between pointer types
    ∗ If sizes are not the same, the least significant bytes are usually preserved

◇ Implicit casts:
  – Compiler tries to perform conversions to match operators and declarations of functions.
  – When the result of an expression does not fit into 32 bits, it is cast to its address.
  – Exception: structures
  – Specific to C, most other languages do not perform this cast

Datatype Conversions

◇ Usually called *casts*. Two main types:
  - A notion of *value* can be preserved
    - integral $\longrightarrow$ real : preserve the value
    - real $\longrightarrow$ integral : truncates away the fractional part
    - small size integral $\longrightarrow$ large size integral : *sign extended*
      - Conversion may take the `floor` or the `ceiling`, depending on the language.
  - Bits can be copied over.
    - Conversion between pointer types
    - If sizes are not the same, the least significant bytes are usually preserved

```
float a;
...
int f(int x) ;
...
x = f(a);
```

a is truncated to int, value is preserved to utmost extent possible

◇ Implicit casts:
  - Compiler tries to perform conversions to match operators and declarations of functions.
  - When the result of an expression does not fit into 32 bits, it is cast to its address.
  - Exception: structures
  - Specific to C, most other languages do not perform this cast

Aggregate Types

- Arrays
- Most langauges provide *records*
  - In C they are called *structures*
  - In object oriented programming they are extended to *objects*
- Unions: specific to C, help save space.
- High-level aggregate datatypes (Python, Ruby):
  - Lists
  - Tuples
  - Sets
  - Dictionaries
  - implemented in libraries for languages without these primitives

C Pointers

- Models the memory address of a datum.

- Memory is an array of bytes (unsigned characters) $\rightarrow$ pointer is an unsigned integer.

- At the type system level, the language distinguishes between pointers and integers for safety reasons.
  - This does not happen in VAL

- Declaration: `type * p`

- Operations:
  - `*p` : dereference
  - `p+k` : pointer arithmetic, points `k*sizeof(*p)` bytes away.

- The address of any *lvalue* can be captured into a pointer with the `&` (address-of) operator.

Pointers to Functions

```
type f() {
  ...
}

...
{
  type (*p)() ; // pointer to a function that returns "type"
  p = &f ; // assign p to address of f
  ...
  (*p)() ;  // calls f
  ...
}
```

Pointers to Functions

```
type f() {
  ...
}

...
{
  type (*p)() ; // pointer to a function that returns "type"
  p = &f ; // assign p to address of f
  ...
  (*p)() ;  // calls f
  ...
}
```

`p=f` would work too

Pointers to Functions

```
type f() {
  ...
}

...
{
  type (*p)() ; // pointer to a function that returns "type"
  p = &f ; // assign p to address of f
  ...
  (*p)() ;  // calls f
  ...
}
```

`p=f` would work too

`f` would be implicitly cast to its address

Pointers to Functions

```
type f() {
  ...
}

...
{
  type (*p)() ; // pointer to a function that returns "type"
  p = &f ; // assign p to address of f
  ...
  (*p)() ;  // calls f
  ...
}
```

`p=f` would work too

`f` would be implicitly cast to its address

`p()` would work too

Pointers to Functions

```
type f() {
  ...
}

...
{
  type (*p)() ; // pointer to a function that returns "type"
  p = &f ; // assign p to address of f
  ...
  (*p)() ;  // calls f
  ...
}
```

`p=f` would work too

`f` would be implicitly cast to its address

`p()` would work too

`p` would be implicitly cast to its dereference

The C++ Reference Type

```
int & x, &y ;
x = 3 ; // allocate space for x and store 3 in the location
x += 2 ; // location pointed to by x holds 5
y = x ; // y is equal to 5
....
{
    int f(int &a, int &b) ;
    f(x,y) ; // x is passed as an address, y is converted to &y
             // both x and y may be changed upon return
    f(2,3) ; // space is allocated for two integers, which are initialized to 2 and 3
             // the addresses of the two integers is passed into f
....
}
....
```

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

`f` is a function that returns a pointer

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function that returns a pointer to a function

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function that returns a pointer to a function that returns a pointer

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function that returns a pointer to a function that returns a pointer to int

Hierarchic Data

Type operators have precedence: () and [] bind tighter than * ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

`a` is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

`a` is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

`a` is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Each layer of pointers must be intialized.

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Drop the a to obtain the cast.

Hierarchic Data

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}
```

Move one * from right side to the left side

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Continue on for each level of pointers

Prolog Terms

- Terms represent tree-like symbolic data
- They are common to symbolic processing languages: Prolog, Ocaml, Haskell, Scheme
- Allow *pattern-matching*: operation that allows extraction of components of syntactic structures.
- In Prolog, it is not possible to specify that the argument is *limited* to a set of terms
- Typed languages, such as Ocaml and Haskell, allow specification of such restrictions.

```
toString(E1+E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"+",S2,")"],S).
toString(E1-E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"-",S2,")"],S).
toString(E1*E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"*",S2,")"],S).
toString(E1/E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"/",S2,")"],S).
toString(X,[Y])   :- atom(X), char_code(X,Y).
```

The Scheme Equivalent

```
(define (toString e)
  (if (pair? e)
      (cond ((eq? (car e) '+) (string-append "(" (toString (cadr e)) "+" (toString (caddr e)) ")" ))
            ((eq? (car e) '-) (string-append "(" (toString (cadr e)) "-" (toString (caddr e)) ")" ))
            ((eq? (car e) '*) (string-append "(" (toString (cadr e)) "*" (toString (caddr e)) ")" ))
            ((eq? (car e) '/) (string-append "(" (toString (cadr e)) "/" (toString (caddr e)) ")" )))
      (symbol->string e)))

(toString '(+ (* a b) (/ c d)))

Return value: "((a*b)+(c/d))"
```

No pattern matching in Scheme!

Dynamic vs. Static Typing

◇ Dynamic Typing (Prolog, Python, Ruby, Javascript):
- Each datum is stored with its type
- Before each operation, the type is checked
  * If cast is possible, then operation proceeds after cast
  * If cast is not possible, the operation fails with error or exception
  * Prolog predicates may just fail with no error message $\rightarrow$ debugging becomes difficult
- Less efficient, due to extra tests at execution time

◇ Static Typing (C, Ocaml, Haskell, Java, C#):
- Types are inferred at compiled time
- Data are stored without type
- If cast is necessary, code for cast is compiled into the executable
- If cast is not possible, compilation error is issued
- More restrictive, since inferring types at compile time is weaker than finding out the types directly during execution.
- More efficient execution, due to lack of type checks at run time.

Terms in Ocaml

```
type expr = Plus of expr * expr         (* means a + b *)
          | Minus of expr * expr        (* means a - b *)
          | Times of expr * expr        (* means a * b *)
          | Divide of expr * expr       (* means a / b *)
          | Value of string             (* "x", "y", "n", etc. *)
  ;;
```

```
let rec to_string e =
  match e with
    Plus (left, right)   -> "(" ^ (to_string left) ^ " + " ^ (to_string right) ^ ")"
  | Minus (left, right)  -> "(" ^ (to_string left) ^ " - " ^ (to_string right) ^ ")"
  | Times (left, right)  -> "(" ^ (to_string left) ^ " * " ^ (to_string right) ^ ")"
  | Divide (left, right) -> "(" ^ (to_string left) ^ " / " ^ (to_string right) ^ ")"
  | Value v -> v
  ;;
```

```
toString(E1+E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"+",S2,")"],S).
toString(E1-E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"-",S2,")"],S).
toString(E1*E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"*",S2,")"],S).
toString(E1/E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"/",S2,")"],S).
toString(X,[Y])   :- atom(X), char_code(X,Y).
```

C Equivalent

```
type foo =  Nothing
          | Int of int
          | Pair of int * int
          | String of string;;
```

```
struct foo {
  int type;
#define TYPE_INT 1
#define TYPE_PAIR_OF_INTS 2
#define TYPE_STRING 3
  union {
    int i;         // If type == TYPE_INT.
    int pair[2];   // If type == TYPE_PAIR_OF_INTS.
    char *str;     // If type == TYPE_STRING.
  } u;
};
```

```
Ocaml                  C                              Prolog

x = Int 1              struct foo x ;                 X = int(1)
                       x.type = TYPE_INT
                       u.i = 1 ;

y = Pair(2,3)          struct foo y ;                 X = pair(2,3)
                       y.type = TYPE_PAIR_OF_INTS
                       u.pair[0] = 2 ;
                       u.pair[1] = 3 ;
```

Terms in Haskell

Data type declaration

```
data Expr = Plus Expr Expr          -- means a + b
          | Minus Expr Expr         -- means a - b
          | Times Expr Expr         -- means a * b
          | Divide Expr Expr        -- means a / b
          | Value String            -- "x", "y", "n", etc.
```

Haskell code

```
toString (Plus left right)   = "(" ++ (toString left) ++ "+" ++ (toString right) ++ ")"
toString (Minus left right)  = "(" ++ (toString left) ++ "-" ++ (toString right) ++ ")"
toString (Times left right)  = "(" ++ (toString left) ++ "*" ++ (toString right) ++ ")"
toString (Divide left right) = "(" ++ (toString left) ++ "/" ++ (toString right) ++ ")"
toString (Value s)           = s
```

Equivalent Prolog code:

```
toString(E1+E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"+",S2,")"],S).
toString(E1-E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"-",S2,")"],S).
toString(E1*E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"*",S2,")"],S).
toString(E1/E2,S) :- !, toString(E1,S1), toString(E2,S2), append(["(",S1,"/",S2,")"],S).
toString(X,[Y])   :- atom(X), char_code(X,Y).
```

Python Datatypes

- ◇ Numeric datatypes as usual + infinite precision numbers
- ◇ High level datatypes as first-class citizens
- ◇ Part of the language, and not a library
- ◇ Convenient syntax, readable programs

Python Lists (1)

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]

>>> a[0]
'spam'

>>> a[3]
1234

>>> a[-2]
100

>>> a[1:-1]
['eggs', 100]

>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]

>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']

>>> a
['spam', 'eggs', 100, 1234]

>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Python Lists (2)

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]

>>> # Remove some:
... a[0:2] = []

>>> a
[123, 1234]

>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']

>>> a
[123, 'bletch', 'xyzzy', 1234]

>>> # Insert (a copy of) itself at the beginning
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]

>>> # Clear the list: replace all items with an empty list
>>> a[:] = []
>>> a
[]

>>> a = ['a', 'b', 'c', 'd']
>>> len(a)
4
```

Python Lists (3)

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3

>>> p[1]
[2, 3]

>>> p[1][0]
2

>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]

>>> q
[2, 3, 'xtra']
```

List Comprehensions

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]

>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]

>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [w.strip() for w in freshfruit]
['banana', 'loganberry', 'passion fruit']

>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]

>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]

>>> mat = [
...         [1, 2, 3],
...         [4, 5, 6],
...         [7, 8, 9],
...        ]

>>> print([[row[i] for row in mat] for i in [0, 1, 2]])
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Python Tuples

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345

>>> t
(12345, 54321, 'hello!')

>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)

>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))

>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0

>>> len(singleton)
1

>>> singleton
('hello',)

>>> x, y, z = t
```

Python Sets

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                       # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}

>>> 'orange' in basket                  # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> a = set('abracadabra') # Demonstrate set operations on unique letters from two words
>>> b = set('alacazam')
>>> a                                   # unique letters in a
{'a', 'r', 'b', 'c', 'd'}

>>> a - b                               # letters in a but not in b
{'r', 'd', 'b'}

>>> a | b                               # letters in either a or b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

>>> a & b                               # letters in both a and b
{'a', 'c'}

>>> a ^ b                               # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}

>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

Python Dictionaries

```
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}

>>> tel['jack']
4098

>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}

>>> list(tel.keys())
['irv', 'guido', 'jack']

>>> sorted(tel.keys())
['guido', 'irv', 'jack']

>>> 'guido' in tel
True
>>> 'jack' not in tel
False

>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}

>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}

>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Looping Techniques (1)

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

```
>>> basket = ['apple',
...           'orange',
...           'apple',
...           'pear',
...           'orange',
...           'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

What Have We Learned

- Python has convenient high-level datatypes
- Data defined by these datatypes have convenient operators that make programs more readable
- Languages who do not have such high-level datatypes as part of the language can add them via libraries and modules.
  - Whether expressiveness is reduced is largely a matter of taste