Multiverse Recursive Descent Grammar Exploration

L. Thomas van Binsbergen

Informatics Institute, University of Amsterdam
ltvanbinsbergen@acm.org

June 21, 2024

My presentation today

I would characterise my presentation today as follows:

.

My presentation today

I would characterise my presentation today as follows:

- Recursive Descent Parsing as a **case study** for Multiverse Debugging.

My presentation today

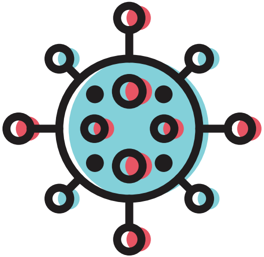
I would characterise my presentation today as follows:

- Recursive Descent Parsing as a **case study** for Multiverse Debugging.
- Helping me get out of my rabbit hole.

My presentation today

I would characterise my presentation today as follows:

- Recursive Descent Parsing as a **case study** for Multiverse Debugging.
- Helping me get out of my rabbit hole.
- A personal confirmation I still nurture a parsing bug.



A General Framework for Multiverse Debugging

Temporal Breakpoints for Multiverse Debugging

Matthias Pasquier
ERTOSGENER
Angers, France
matthias.pasquier@ertosgener.com

Ciprian Teodorov
Lab-STICC CNRS UMR 6285
ENSTA Bretagne
Brest, France
ciprian.teodorov@ensta-bretagne.fr

Frédéric Jouault
LERIA, University of Angers
ESEO
Angers, France
frederic.jouault@eseo.fr

Matthias Brun
LERIA, University of Angers
ESEO
Angers, France
matthias.brun@eseo.fr

Luka Le Roux
Lab-STICC CNRS UMR 6285
ENSTA Bretagne
Brest, France
luka.le_roux@ensta-bretagne.fr

Loïc Lagadec
Lab-STICC CNRS UMR 6285
ENSTA Bretagne
Brest, France
loic.lagadec@ensta-bretagne.fr

Figure: DOI: `https://doi.org/10.1145/3623476.3623526`

Language-parameteric debugging framework for non-deterministic/concurrent executions.

Interactive execution models

Incremental programming: build towards a larger program by submitting program fragments one-by-one and receive immediate feedback. E.g., Python, Jupyter, REPLs in general..

Interactive execution models

Incremental programming: build towards a larger program by submitting program fragments one-by-one and receive immediate feedback. E.g., Python, Jupyter, REPLs in general..

$\hookrightarrow$ compare with *Stepwise debugging*: step through the execution of a program to observe evolution of state and effects, halting at selected ‘breakpoints’

Interactive execution models

Incremental programming: build towards a larger program by submitting program fragments one-by-one and receive immediate feedback. E.g., Python, Jupyter, REPLs in general..

$\hookrightarrow$ compare with *Stepwise debugging*: step through the execution of a program to observe evolution of state and effects, halting at selected 'breakpoints'

Exploratory programming: build towards a larger program by attempting, comparing and revisiting alternative extensions

Interactive execution models

Incremental programming: build towards a larger program by submitting program fragments one-by-one and receive immediate feedback. E.g., Python, Jupyter, REPLs in general..

↪ compare with *Stepwise debugging*: step through the execution of a program to observe evolution of state and effects, halting at selected 'breakpoints'

Exploratory programming: build towards a larger program by attempting, comparing and revisiting alternative extensions

↪ *Omniscient/Back-in-time debugging*:
record and revisit previous, selected execution points to observe state and effects

Interactive execution models

Incremental programming: build towards a larger program by submitting program fragments one-by-one and receive immediate feedback. E.g., Python, Jupyter, REPLs in general..

$\hookrightarrow$ compare with *Stepwise debugging*: step through the execution of a program to observe evolution of state and effects, halting at selected 'breakpoints'

Exploratory programming: build towards a larger program by attempting, comparing and revisiting alternative extensions

$\hookrightarrow$ *Omniscient/Back-in-time debugging*:
record and revisit previous, selected execution points to observe state and effects

$\hookrightarrow$ *Multiverse debugging*: (exhaustive) exploration of concurrent executions
(with state reductions and temporal breakpoints)

Interactive programming models

Commonality

A transition relation $\mathit{state} \xrightarrow{\mathit{action}} \mathit{state}'$ describes a tree of executions and reachable states.

A Principled Approach to REPL Interpreters

L. Thomas van Binsbergen
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
ltvanbinsbergen@acm.org

Mauricio Verano Merino
Eindhoven University of Technology
Eindhoven, The Netherlands
m.verano.merino@tue.nl

Pierre Jeanjean
Inria, University of Rennes, CRNS, IRISA
Rennes, France
pierre.jeanjean@inria.fr

Tijs van der Storm
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
University of Groningen
Groningen, The Netherlands
storm@cwi.nl

Benoit Combemale
University of Rennes, Inria, CNRS, IRISA
Rennes, France
benoit.combemale@irit.fr

Olivier Barais
University of Rennes, Inria, CNRS, IRISA
Rennes, France
olivier.barais@irisa.fr

Figure: doi: `https://doi.org/10.1145/3410257`

Context-free grammars and non-determinism

Observation: The generation of a sentence by a grammar can be seen as a particular execution of a non-deterministic program.

Context-free grammars and non-determinism

Observation: The generation of a sentence by a grammar can be seen as a particular execution of a non-deterministic program.

Similarly: A complete, generalised parser attempts to find all executions that generate a particular sentence (the input sentence of the parser).

Context-free grammars and non-determinism

Observation: The generation of a sentence by a grammar can be seen as a particular execution of a non-deterministic program.

Similarly: A complete, generalised parser attempts to find all executions that generate a particular sentence (the input sentence of the parser).

Question: In what ways can an omniscient, exploratory, multiverse debugger for grammars assist the grammar engineering process?

Context-free grammars and non-determinism

Observation: The generation of a sentence by a grammar can be seen as a particular execution of a non-deterministic program.

Similarly: A complete, generalised parser attempts to find all executions that generate a particular sentence (the input sentence of the parser).

Question: In what ways can an omniscient, exploratory, multiverse debugger for grammars assist the grammar engineering process?

Today

- A theoretical **framework** for multiverse grammar exploration based on the execution threads encountered in recursive-descent parsers
- A prototype implementation of a **tool** that supports sentence generation, deterministic parsing, error-recovery, deterministic error diagnosis, ...
- TODO: error diagnosis in complete parsing, **evaluation**: *does anyone want this*?

Syntax Analysis – conventional

Definition

A *grammar* G is a set of productions of the form (X, α) with:

- $X \in N$, referred to as the left-hand side, with N the set of nonterminals
- $\alpha \in (N \cup T)^*$, referred to as the right-hand side
- $T \cap N = \emptyset$, with T the set of terminals

Syntax Analysis – conventional

Definition

A *grammar* G is a set of productions of the form (X, α) with:

- $X \in N$, referred to as the left-hand side, with N the set of nonterminals
- $\alpha \in (N \cup T)^*$, referred to as the right-hand side
- $T \cap N = \emptyset$, with T the set of terminals

Definition

The relations $\alpha \to \beta$ (derivation steps) and $\alpha \dashrightarrow \tau$ (left-most derivations) are defined as the smallest set such that:

- $X \to \alpha$ if and only if $(X, \alpha) \in G$
- $\tau X \alpha \to \tau \beta \alpha$ if and only if $X \to \beta$ and $\tau \in T(G)^*$ (the terminal symbols occuring in G)
- $\alpha_0 \dashrightarrow \tau$ if and only if there is a sequence $\alpha_0 \to \alpha_1, \alpha_1 \to \alpha_2, \ldots, \alpha_{n-1} \to \tau$ with $n \geq 0$

Example – sentence generation

Definition

The language $L(G)$ described by grammar G is the set of all sentences τ with $\mathcal{X} \dashrightarrow \tau$ (and $\mathcal{X} \in N$ a nominated start nonterminal).

$$\mathcal{X} \to A\ \mathbf{;}\ \mid\ B$$
$$A \to a\ A\ \mid\ \epsilon$$
$$B \to B\ b\ \mid\ \epsilon$$

Example – sentence generation

Definition

The language $L(G)$ described by grammar G is the set of all sentences τ with $\mathcal{X} \dashrightarrow \tau$ (and $\mathcal{X} \in N$ a nominated start nonterminal).

$$\mathcal{X} \to A\ \mathbf{;}\ \mid\ B$$
$$A \to a\ A\ \mid\ \epsilon$$
$$B \to B\ b\ \mid\ \epsilon$$

$$\begin{aligned} \mathcal{X} &\to A; \\ &\to aA; \\ &\to aaA; \\ &\to aaaA; \\ &\to aaa; \end{aligned}$$

Example – sentence generation

Definition

The language $L(G)$ described by grammar G is the set of all sentences τ with $\mathcal{X} \dashrightarrow \tau$ (and $\mathcal{X} \in N$ a nominated start nonterminal).

$$\begin{aligned} \mathcal{X} &\to A\,\mathbf{;} \mid B \\ A &\to a\ A \mid \epsilon \\ B &\to B\ b \mid \epsilon \end{aligned}$$

$$\begin{aligned} \mathcal{X} &\to A; \\ &\to aA; \\ &\to aaA; \\ &\to aaaA; \\ &\to aaa; \end{aligned}$$

$$\begin{aligned} \mathcal{X} &\to B \\ &\to Bb \\ &\to b \end{aligned}$$

Example – sentence generation

Definition

The language $L(G)$ described by grammar G is the set of all sentences τ with $\mathcal{X} \dashrightarrow \tau$ (and $\mathcal{X} \in N$ a nominated start nonterminal).

$$\begin{aligned} \mathcal{X} &\rightarrow A\,\mathbf{;} \mid B \\ A &\rightarrow a\ A \mid \epsilon \\ B &\rightarrow B\ b \mid \epsilon \end{aligned}$$

$$\begin{aligned} \mathcal{X} &\rightarrow A; \\ &\rightarrow aA; \\ &\rightarrow aaA; \\ &\rightarrow aaaA; \\ &\rightarrow aaa; \end{aligned}$$

$$\begin{aligned} \mathcal{X} &\rightarrow B \\ &\rightarrow Bb \\ &\rightarrow b \end{aligned}$$

Recursive-Descent Parsing: choose alternates based on *next* token expectation from input

Example – Recursive Descent Parsing (RDP)

$$\mathcal{X} \to A \text{ ; } \mid B$$
$$A \to a\ A \mid \epsilon$$
$$B \to B\ b \mid \epsilon$$

Input: [ ] [b ,b]
Next action: **descend**$(\mathcal{X}, B)$

Stack: [$(\mathcal{X}' \to \cdot\mathcal{X}, 0)$]

Example – Recursive Descent Parsing (RDP)

$$\mathcal{X} \to A\ \mathbf{;}\ \mid\ B$$
$$A \to a\ A\ \mid\ \epsilon$$
$$B \to B\ b\ \mid\ \epsilon$$

Input: [ ] [b ,b]
Next action: **descend**(B, Bb)

Stack: [$(\mathcal{X}' \to \cdot\mathcal{X}, 0)$, $(\mathcal{X} \to \cdot B, 0)$]

Example – Recursive Descent Parsing (RDP)

$$\mathcal{X} \to A\ \mathbf{;}\ \mid\ B$$
$$A \to a\ A\ \mid\ \epsilon$$
$$B \to B\ b\ \mid\ \epsilon$$

Input: [ ] [b ,b]
Next action: **descend**(B, Bb)

Stack: [$(\mathcal{X}' \to \cdot\mathcal{X}, 0)$, $(\mathcal{X} \to \cdot B, 0)$, $(B \to \cdot Bb, 0)$]

Example – Recursive Descent Parsing (RDP)

$$\mathcal{X} \to A \, \textbf{;} \mid B$$
$$A \to a\, A \mid \epsilon$$
$$B \to B\, b \mid \epsilon$$

Input: [ ] [b ,b]
Next action: $\textbf{descend}(B, \epsilon)$

Stack: [$(\mathcal{X}' \to \cdot\mathcal{X}, 0)$, $(\mathcal{X} \to \cdot B, 0)$, $(B \to \cdot Bb, 0)$, $(B \to \cdot Bb, 0)$]

Example – Recursive Descent Parsing (RDP)

$$\mathcal{X} \to A\ \mathbf{;}\ \mid\ B$$
$$A \to a\ A \mid \epsilon$$
$$B \to B\ b \mid \epsilon$$

Input: [ ] [b ,b]
Next action: **ascend**

Stack: [$(\mathcal{X}' \to \cdot\mathcal{X}, 0)$, $(\mathcal{X} \to \cdot B, 0)$, $(B \to \cdot Bb, 0)$, $(B \to \cdot Bb, 0)$, $(B \to \cdot, 0)$]

Example – Recursive Descent Parsing (RDP)

$$\mathcal{X} \rightarrow A\ \textbf{;} \mid B$$
$$A \rightarrow a\ A \mid \epsilon$$
$$B \rightarrow B\ b \mid \epsilon$$

Input: [ ] [b ,b]
Next action: **match**(b)

Stack: [$(\mathcal{X}' \rightarrow \cdot\mathcal{X}, 0)$, $(\mathcal{X} \rightarrow \cdot B, 0)$, $(B \rightarrow \cdot Bb, 0)$, $(B \rightarrow B \cdot b, 0)$]

Example – Recursive Descent Parsing (RDP)

$$\mathcal{X} \rightarrow A\ \mathbf{;}\ \mid\ B$$
$$A \rightarrow a\ A\ \mid\ \epsilon$$
$$B \rightarrow B\ b\ \mid\ \epsilon$$

Input: [b] [b]
Next action: **ascend**

Stack: [$(\mathcal{X}' \rightarrow \cdot\mathcal{X}, 0)$, $(\mathcal{X} \rightarrow \cdot B, 0)$, $(B \rightarrow \cdot Bb, 0)$, $(B \rightarrow Bb\cdot, 0)$]

Example – Recursive Descent Parsing (RDP)

$\mathcal{X} \to A \,\mathbf{;} \mid B$

$A \to a\ A \mid \epsilon$

$B \to B\ b \mid \epsilon$

Input: [b] [b]

Next action: **match**(b)

Stack: [$(\mathcal{X}' \to \cdot\mathcal{X}, 0)$, $(\mathcal{X} \to \cdot B, 0)$, $(B \to B \cdot b, 0)$]

Example – Recursive Descent Parsing (RDP)

$$\mathcal{X} \to A\ \mathbf{;}\ \mid\ B$$
$$A \to a\ A \mid \epsilon$$
$$B \to B\ b \mid \epsilon$$

Input: [b, b] [ ]
Next action: **ascend**

Stack: [$(\mathcal{X}' \to \cdot\mathcal{X}, 0)$, $(\mathcal{X} \to \cdot B, 0)$, $(B \to Bb\cdot, 0)$]

Example – Recursive Descent Parsing (RDP)

$$\mathcal{X} \to A\ \textbf{;}\ \mid\ B$$
$$A \to a\ A \mid \epsilon$$
$$B \to B\ b \mid \epsilon$$

Input: [b, b] [ ]
Next action: **ascend**

Stack: [$(\mathcal{X}' \to \cdot\mathcal{X}, 0)$, $(\mathcal{X} \to B\cdot, 0)$]

Example – Recursive Descent Parsing (RDP)

$$\mathcal{X} \rightarrow A \text{ ; } \mid B$$
$$A \rightarrow a\ A \mid \epsilon$$
$$B \rightarrow B\ b \mid \epsilon$$

Input: [b, b] [ ]
Next action: **accept**

Stack: [$(\mathcal{X}' \rightarrow \mathcal{X}\cdot, 0)$]

Example – Recursive Descent Parsing (RDP)

$$\mathcal{X} \rightarrow A\ \mathbf{;}\ \mid\ B$$
$$A \rightarrow a\ A\ \mid\ \epsilon$$
$$B \rightarrow B\ b\ \mid\ \epsilon$$

Input: [b, b] [ ]
Next action: **accept**

Stack: [$(\mathcal{X}' \rightarrow \mathcal{X}\cdot, 0)$]

Common problems in RD-Parsing

- Left-recursion: choosing the same alternative without progress
- Non-predictive: alternates have (indirect) common prefixes
- Non-determinism cannot be avoid in general case (without solving the parse problem)

Syntax Analysis – Alternative

Definition – Syntax of Actions

$$a : action ::= \textbf{match}(t) \mid \textbf{descend}(X, \alpha) \mid \textbf{ascend} \mid \textbf{accept}$$

Constraints: $t \in T, X \in N, \alpha \in (T \cup N)^*$

Syntax Analysis – Alternative

Definition – Syntax of Actions

$$a : action ::= \textbf{match}(t) \mid \textbf{descend}(X, \alpha) \mid \textbf{ascend} \mid \textbf{accept}$$

Constraints: $t \in T, X \in N, \alpha \in (T \cup N)^*$

Definition – Semantics of Actions

A configuration γ is a structure $\langle S, i \rangle$ with:

- S a call-stack; a sequence of *items* denoted $[(X \to \alpha \cdot \beta, k), S']$ with $k \geq 0$
- $i \geq 0$ an index into some input sentence $\mathcal{I}$.

The semantics of actions is captured by the transition relation $\gamma \xrightarrow{a} \gamma'$.

A (complete) parsing *thread* is a (longest) sequence $\langle [(\mathcal{X}' \to \cdot\mathcal{X}, 0)], 0 \rangle \xrightarrow{a_0} \ldots \xrightarrow{a_n} \langle S, m \rangle$ (for some m and $\mathcal{X}' \in N, \mathcal{X}' \notin N(G)$). A thread is successful if $S = []$.

Evolution of RDP Programs in transition system Γ_S

$$\frac{I_i = t \quad s = t}{\langle[(X \to \alpha \cdot s\beta, k), S], i\rangle \xrightarrow{match(t)} \langle[(X \to \alpha t \cdot \beta, k), S], i+1\rangle}\ \text{MATCH}$$

$$\frac{s = X \quad (X, \delta) \in G}{\langle[(Y \to \alpha \cdot s\beta, k), S], i\rangle \xrightarrow{descend(X,\delta)} \langle[(X \to \cdot\delta, i), (Y \to \alpha \cdot X\beta, k), S], i\rangle}\ \text{DESCEND}$$

$$\frac{[(Y \to \alpha \cdot X\beta, k'), S'] = S}{\langle[(X \to \alpha\cdot, k), S], i\rangle \xrightarrow{ascend} \langle[(Y \to \alpha X \cdot \beta, k'), S'], i\rangle}\ \text{ASCEND}$$

$$\frac{|\mathcal{I}| = i}{\langle[(\mathcal{X}' \to \mathcal{X}\cdot, 0)], i\rangle \xrightarrow{accept} \langle[], i\rangle}\ \text{ACCEPT}$$

Evolution of RDP Programs in transition system Γ_G

$$\frac{s = t}{\langle[(X \to \alpha \cdot s\beta, k), S], i\rangle \xrightarrow{match(t)} \langle[(X \to \alpha t \cdot \beta, k), S], i+1\rangle}\ \textsc{Match}$$

$$\frac{s = X \quad (X, \delta) \in G}{\langle[(Y \to \alpha \cdot s\beta, k), S], i\rangle \xrightarrow{descend(X,\delta)} \langle[(X \to \cdot\delta, i), (Y \to \alpha \cdot X\beta, k), S], i\rangle}\ \textsc{Descend}$$

$$\frac{[(Y \to \alpha \cdot X\beta, k'), S'] = S}{\langle[(X \to \alpha\cdot, k), S], i\rangle \xrightarrow{ascend} \langle[(Y \to \alpha X \cdot \beta, k'), S'], i\rangle}\ \textsc{Ascend}$$

$$\frac{}{\langle[(\mathcal{X}' \to \mathcal{X}\cdot, 0)], i\rangle \xrightarrow{accept} \langle[], i\rangle}\ \textsc{Accept}$$

Demo 1 – guided sentence generation

Syntax Analysis – Claims

For every **derivation** *there is exactly one successful thread in* Γ_G *and vice versa.*

Syntax Analysis – Claims

For every **derivation** *there is exactly one successful thread in* Γ_G *and vice versa.*

For every **parse tree** *of* $\mathcal{I}$ *there is exactly one successful thread in* Γ_S *and vice versa.*

Syntax Analysis – Claims

For every **derivation** *there is exactly one successful thread in* Γ_G *and vice versa.*

For every **parse tree** *of* $\mathcal{I}$ *there is exactly one successful thread in* Γ_S *and vice versa.*

A **parser** *synthesises a successful thread for a sentence* $\mathcal{I}$ *if there is one.*

Syntax Analysis – Claims

For every **derivation** *there is exactly one successful thread in* Γ_G *and vice versa.*

For every **parse tree** *of* $\mathcal{I}$ *there is exactly one successful thread in* Γ_S *and vice versa.*

A **parser** *synthesises a successful thread for a sentence* $\mathcal{I}$ *if there is one.*

A **complete parser** *synthesises all successful threads for a sentence* $\mathcal{I}$*.*

Syntax Analysis – Claims

For every **derivation** *there is exactly one successful thread in* Γ_G *and vice versa.*

For every **parse tree** *of* $\mathcal{I}$ *there is exactly one successful thread in* Γ_S *and vice versa.*

A **parser** *synthesises a successful thread for a sentence* $\mathcal{I}$ *if there is one.*

A **complete parser** *synthesises all successful threads for a sentence* $\mathcal{I}$*.*

A **general parser** *does the above for any context-free grammars (no restrictions).*

Common solutions to problems with RD-Parsing

Recall: Common problems in RD-Parsing

- Left-recursion: choosing the same alternative without progress
- Non-predictive: alternates have (indirect) common prefixes
- Non-determinism cannot be avoid in general case (without solving the parse problem)

Common solutions to pruning undesirable paths

- Left-recursion removal: preserves language, modifies set of threads
- Alternative: make at most $|\mathcal{I}|$ recursive calls
- Left-factoring: preserves language, modifies set of threads
- $k-$lookahead: sound, reduces set of threads
- Accepting only predictive, $LL(k)-$grammars: not general

Lookahead Strategies Re-imagined

Defining – Language-Preserving Pruning Strategies

A *pruning strategy* defines additional conditions (on top of Γ_G), removing transitions $\gamma \xrightarrow{a} \gamma'$.

A pruning strategy is *language preserving* if it reduces the set of threads whilst preserving, for every sentence in the language, at least one successful thread yielding that sentence.

A pruning strategy is *sound* if it reduces the set of threads without removing successful threads.

Note: left-biased choice not language-preserving, lookahead is.

Lookahead Strategies Re-imagined

Defining – Language-Preserving Pruning Strategies

A *pruning strategy* defines additional conditions (on top of Γ_G), removing transitions $\gamma \xrightarrow{a} \gamma'$.

A pruning strategy is *language preserving* if it reduces the set of threads whilst preserving, for every sentence in the language, at least one successful thread yielding that sentence.

A pruning strategy is *sound* if it reduces the set of threads without removing successful threads.

Note: left-biased choice not language-preserving, lookahead is.

Definition – Ambiguity Reduction Strategies

A *perfect disambiguation strategy* is a language-preserving pruning strategy that preserves exactly one successful thread for every sentence in the language.

Note: many practical disambiguation strategies are not perfect (e.g., follow-restriction)

Pruning Examples

Definition – one token lookahead pruning

Remove $\langle S, k\rangle \xrightarrow{\textbf{descend}(X,\alpha)} \gamma'$ for any S, k, γ', X, α if it holds that every shortest sub-thread $\gamma' \xrightarrow{a_1} \ldots \xrightarrow{a_n} \gamma_n$ that ends with

- $a_n = \textbf{match}(t)$ has $t \neq \mathcal{I}_k$, or
- $\gamma_n = \langle[(\mathcal{X}' \rightarrow \mathcal{X}\cdot, 0)], k'\rangle$ has $k' \neq\mid \mathcal{I} \mid$.

Pruning Examples

Definition – one token lookahead pruning

Remove $\langle S, k\rangle \xrightarrow{\textbf{descend}(X,\alpha)} \gamma'$ for any S, k, γ', X, α if it holds that every shortest sub-thread $\gamma' \xrightarrow{a_1} \ldots \xrightarrow{a_n} \gamma_n$ that ends with

- $a_n = \textbf{match}(t)$ has $t \neq \mathcal{I}_k$, or
- $\gamma_n = \langle[(\mathcal{X}' \to \mathcal{X}\cdot, 0)], k'\rangle$ has $k' \neq \mid \mathcal{I} \mid$.

Definition – left-recursion termination

Remove $\langle S, k\rangle \xrightarrow{\textbf{descend}(X,\alpha)} \gamma'$ for any S, k, γ', X, α when S has $\mid \mathcal{I} \mid$ or more items of the form $(Y \to \ldots \cdot X \ldots, k)$ for any Y.

i.e., $\mid \mathcal{I} \mid$ descend actions have already been performed on X without progress.
Note: this pruning strategy is unsound for some grammars.

Action-lookahead

Definition – k-actions lookahead

Remove $\gamma_0 \xrightarrow{\textbf{descend}(X,\alpha)} \gamma_1$ if all (subsequent) threads of k actions or less cannot transition (considering any collection of additional pruning strategies).

With this strategy, a descend action is only possible if all subsequent threads are of a length $i > k$ or at least one successful thread has been identified.

Demo 2

Demo 2 with the different pruning strategies applied.

Deterministic parse error diagnosis

The parse error diagnosis problem:

> *Given a stack-trace and an input sentence location, answer the following: Did I expect to be at a different grammar location or a different input sentence location? Does the grammar or input require modification?*

Deterministic parse error diagnosis

The parse error diagnosis problem:

Given a stack-trace and an input sentence location, answer the following: Did I expect to be at a different grammar location or a different input sentence location? Does the grammar or input require modification?

Suggested protocol for using RDP-Debugging tool:

1. Configure tool to use same lookahead strategy as employed parser
2. Run on grammar and input until parse error.
3. Observe stack-trace and input location.
4. Weaken input sentence constraints (move from Γ_S to Γ_G semantics).
5. Experiment with additional or dropped tokens (error recovery, next slide).
6. Jump back to last transition that still matched expectation (omniscient debugging).
7. Weaken the applied set of lookahead/pruning strategies.
8. Explore alternative paths to better understand divergence from expectation.

Error Recovery – Additional Rules

Definition – Additional Syntax of Actions

$$a : action ::= \ldots \mid \textbf{drop}$$

Additional Rules for a System Γ_{err}

$$\frac{s = t}{\langle [(X \to \alpha \cdot s\beta, k), S], i \rangle \xrightarrow{match(t)} \langle [(X \to \alpha t \cdot \beta, k), S], i \rangle} \textsc{Match-Any}$$

$$\frac{}{\langle S, i \rangle \xrightarrow{drop} \langle S, i + 1 \rangle} \textsc{Drop}$$

Non-deterministic parse error diagnosis?

Observation: Deterministic parsing is akin to finding *any* state which errors or accepts (assuming lookahead is sufficient to cancel all non-determinism).

Observation: Non-deterministic parsing akin to finding *all* states that error or accept.

Problem: due to non-determinism and ambiguity, parser can find many error threads!
(possibly infinitely or exponentially many)

Question: how can multiverse debugging be used for diagnosing errors in complete parsing?

Conclusion

Where I am today

- A theoretical **framework** for multiverse grammar exploration based on the execution threads encountered in recursive-descent parsers
- A prototype implementation of a **tool** that supports sentence generation, deterministic parsing, error-recovery, deterministic error diagnosis, ...
- The underlying reachability graph can be modified on the fly by the user
- TODO: error diagnosis in complete parsing, **evaluation**: *does anyone want this*?

Case study conclusions

- Pruning strategies can be applied in other case studies to reduce search space
- Not all actions available to the user should be available to the breakpoint finder
- We'd like to have a `find all <breakpoint>` command (for non-deterministic parsing)

Multiverse Recursive Descent Grammar Exploration

L. Thomas van Binsbergen

Informatics Institute, University of Amsterdam
ltvanbinsbergen@acm.org

June 21, 2024

Image Credits

Man digging a hole and virus images by https://www.vectorportal.com.