Санкт-Петербургский Государственный
Университет
Математико-Механический факультет

Кафедра системного программирования

Восстановление текстов программ по преобразованному синтаксическому дереву

Дипломная работа студента 545 группы

Литвинова Ю.В.

Научный руководитель	..... /подпись/	асп. Д.В. Копаев
Рецензент	..... /подпись/	ст. преп. Я.А. Кириленко
“Допустить к защите” зав. кафедры	..... /подпись/	д.ф.-м.н. А.Н. Терехов

Санкт-Петербург
2007

Содержание

1 Введение

Данная работа создавалась в рамках системы автоматического реинжиниринга программных комплексов Relativity Modernization Workbench [2]. Задачи реинжиниринга возникают, когда продолжать поддержку существующей программы становится слишком дорого, а полностью отказаться от неё невозможно. Довольно часто программы являются основным, а иногда даже единственным хранилищем информации о процессах организации, но вносить в них изменения и исправлять ошибки становится со временем все сложнее. Такие программы могут использоваться десятилетиями, за это время не только их разработчики, но и специалисты, знакомые с использованными при создании этих программ технологиями, становятся недоступны.

В таком случае может помочь реинжиниринг. К задачам реинжиниринга, помимо перевода программ на новые языки программирования, технологии и аппаратные платформы, относится исследование старых программ с целью выявить заложенную в них логику, относящуюся к бизнес-процессам, а также преобразования программ и приведение их к виду, более пригодному для сопровождения.

Оказалось, что немногие организации готовы пойти на риск и перейти на новые технологии, автоматически сгенерировав программу на новом языке по старой программе, несмотря на то, что существующие средства позволяют достаточно хорошо решить эту задачу. Вместо этого, они предпочитают продолжать сопровождение старых программ, пользуясь средствами реинжиниринга для анализа, автоматической реструктуризации, удаления мёртвого кода, различных оптимизаций и рефакторингов программ.

В данной ситуации возникает задача генерации кода программы на исходном языке. Обычно преобразования приводят к изменению относительно небольших фрагментов программ (например, переименование переменных, протягивание констант, создание подпрограммы из заданного фрагмента кода), поэтому естественным желанием было бы максимально использовать текст исходной программы, внося в него изменения лишь в тех местах, где это действительно необходимо. Желательно, чтобы пользователь воспринимал сгенерированную программу как оригинальную, но с внесёнными изменениями, а не как что-то новое. Вместе с тем, программа после таких преобразований должна быть синтаксически корректна и удобна для чтения. Это означает, что вновь порождённый текст должен быть некоторым естественным образом соединён с существующим, словно изменения вносились редактированием текста.

В Modernization Workbench существовал алгоритм генерации, однако,

он имел ряд недостатков. Целью данной работы было создание нового алгоритма, решающего задачу минимизации изменений, вносимых в исходный код при перегенерации программы. Алгоритм предполагалось использовать для генерации текстов программ для языков COBOL [5] и RPG [7].

В части 2 приводится обзор существующих методов генерации текста по дереву программы, в части 3 описываются возможные преобразования программы, представление информации о связи дерева и исходного файла, особенности порождения текста при тех или иных изменениях, какие трудности при этом возникают и что необходимо сделать для того, чтобы достичь поставленной цели. В части 4 даётся обзор предыдущего алгоритма и его недостатков, из-за которых возникла необходимость в создании нового алгоритма. В части 5 описывается предлагаемое решение, в части 6 описаны достоинства и недостатки нового решения по сравнению со старым, в части 7 даётся краткий обзор достигнутых результатов, в приложениях приведены примеры работы старого и нового алгоритма в различных трудных ситуациях.

2 Обзор существующих технологий

Тексты программ могут порождаться только по информации, сохранённой в синтаксическом дереве, или с использованием текста из оригинального файла. Второй подход специфичен для задач реинжиниринга, поскольку позволяет сохранить пользовательское форматирование, первый подход широко используется для различных задач, где требуется порождение текста, и хорошо освещён в литературе.

Обычно генерация без использования исходного текста не представляется сложной, достоин упоминания случай генерации арифметических выражений с учётом приоритета операторов, где требуется минимизировать количество скобок [8]. Кроме того, существуют методы автоматической генерации генераторов текста вместе с синтаксическими анализаторами по грамматике языка [9]. В статье упоминается различие между понятиями "дерево разбора" и "абстрактное синтаксическое дерево". Сгенерировать программу по дереву разбора не представляет труда, так как информация о ключевых словах, приоритете и ассоциативности операторов и т.д. доступна явно. Обычно синтаксические анализаторы выдают абстрактное синтаксическое дерево, которое содержит значительно меньше информации, поэтому требуется определять синтаксические правила для каждого узла дерева.

Такой подход к генерации текстов используется в том числе и для задач преобразования программ, например, в TXL [6], где результат преобразования печатается по дереву порождённым из спецификации языка генератором. В DMS [3] применяется похожий подход, по спецификации языка порождается генератор, который может быть использован для форматирования текста программы, или наоборот, для запутывания исходного кода. В Modernization Workbench текст строится по синтаксическому дереву, если целевой язык отличается от исходного [1]. Кроме того, печать по дереву широко используется для генерации текстов по различным спецификациям и для сериализации данных в формате, удобном для чтения человеком [4].

Ни один из рассмотренных подходов не применяет генерацию с использованием оригинального текста. Возможно, это связано со специфичностью задачи минимизации изменений в тексте программы, ведь для таких задач, как перевод программы на новый язык программирования или форматирование текста минимизация изменений не требуется.

3 Постановка задачи

3.1 Возможные преобразования дерева

Результат преобразования старой программы называется слайсом (“срез” программы). Для языка COBOL в Modernization Workbench существует генерация следующих видов слайсов:

Structure-based — выделение заданной части кода в отдельную программу, вставка вызовов вновь сгенерированной программы в исходную.

Computation-based — выделение кода, необходимого для вычисления значения заданной переменной в заданной точке программы.

Domain-based — специализация программы на основе значений заданных переменных. Имеется возможность зафиксировать значения (или набор значений) переменных или задать их в определённом месте программы.

Event Injection — вставка операций приёма и посылки сообщений, для облегчения перехода на событийно-ориентированную архитектуру.

Dead Code Elimination — удаление недостижимых операторов и неиспользуемых элементов данных. Может генерироваться отдельно, может входить как опция в другие виды слайсов.

Entry Point Isolation — выделение операторов, достижимых из заданной точки входа, в отдельную программу.

Для языка RPG поддерживаются только Dead Code Elimination и Structure-based слайсы.

Отличительные особенности рассматриваемых языков, осложняющие задачу восстановления исходных текстов, включают активное использование препроцессора, наличие большого числа (более 10) диалектов языка COBOL, имеющих различающийся синтаксис многих конструкций, двумерный синтаксис языка RPG (т.е. синтаксическое значение лексемы, как правило, зависит от её положения в строке).

Генерация слайса в Modernization Workbench происходит следующим образом: сначала исходная программа проходит верификацию (синтаксический анализ и сбор различной информации, необходимой для работы интерактивных инструментов), затем (после того, как пользователь указал тип и параметры слайса) выполняется преобразующий проход, после

чего запускается генерация, которая должна преобразовать модифицированное синтаксическое дерево в текст программы.

3.2 Связь дерева и исходного текста

Для задания соответствия между фрагментами текста исходной программы и узлами в синтаксическом дереве используется привязка — для узла задаётся номер файла, номер строки и колонки начала и конца текстового представления узла. Узлы, которых не было в тексте программы (например, добавленные синтаксическим анализатором вспомогательные узлы или узлы, добавленные преобразующим проходом), привязки не имеют. Синтаксическое дерево не содержит в себе информации о каждой лексеме из текста, т.е., например, выражение из языка COBOL

```
 PROCEDURE DIVISION
```

будет представлено в дереве одним узлом, но синтаксически корректной является и такая запись:

```
 PROCEDURE
* Comment
 DIVISION
```

Узел, соответствующий конструкции PROCEDURE DIVISION, будет включать в свою привязку текст другого узла — комментария. Понятно, что такая особенность представления информации о привязке осложняет работу генерации, но полное дерево разбора было бы чрезвычайно неудобно для работы преобразующих проходов, поскольку содержало бы массу узлов, необходимых только для задания привязки.

3.3 Описание преобразований

Для описания преобразований, произведённых над деревом, используются атрибуты. Для того, чтобы определить, кем и с какой целью был создан данный узел, узел помечается специальным флагом — креатором (от англ. Creator). Генерация может использовать следующие атрибуты узлов:

Deleted — узел удалён в результате преобразований. В зависимости от настроек слайса, текст такого узла не должен быть распечатан или должен быть распечатан как комментарий. Если узел создан преобразующими проходами и помечен этим атрибутом, он не должен печататься.

Modified — узел изменён. Выставляется, если изменились поля узла, добавлены или удалены печатаемые сыновья, и вообще, в любом случае, когда текстовое представление узла изменилось, и его нельзя распечатать по привязке, используя текст из исходного файла. Такой узел должен быть распечатан по описанию синтаксиса языка с использованием информации, сохранённой в дереве (далее мы будем употреблять термин "распечатан по дереву" для таких случаев). В случае, если задана соответствующая опция генерации, такой узел должен быть помечен комментарием.

Moved — узел перемещён. Выставляется, если узел изменил родителя или свою позицию в синтаксическом представлении родителя, т.е. его привязка больше не соответствует его положению относительно других узлов. Узел в таком случае должен быть распечатан по привязке, но его местоположение в выходном файле должно определяться описанием синтаксиса языка, а не привязкой узла.

BadlyConnected — узел плохо привязан. Выставляется, если привязка узла известна только с точностью до файла. Обычно выставляется синтаксическим анализатором при раскрытии директив препроцессора и обработке прочих конструкций, точное положение которых в файле неизвестно на этапе синтаксического анализа. Генерация должна распечатать такой узел по дереву в тот файл, в котором был текст этого узла в оригинальной программе.

Сброшенная привязка — не реализована как атрибут, а является неким особым значением привязки. Означает, что текста узла не было в тексте программы. Такой узел должен быть распечатан по дереву, в позиции, определённой синтаксическим правилом родителя.

Креаторы:

PARSER — узел, прочитанный синтаксическим анализатором из исходного файла или созданный синтаксическим анализатором для структурирования дерева. Не обязательно имеет текстовый прообраз в исходном файле, но в этом случае синтаксис допускает отсутствие текста для этого узла. Может иметь сброшенную привязку, однако, если привязка известна, она включает весь текст, относящийся к узлу (может включать и другой текст).

PARSER_AUX — вспомогательный узел, не имеющий соответствующего ему текста в исходном файле. Создаётся синтаксическим ана-

лизатором для использования в преобразующих проходах, для сбора информации и т.д. Текст такого узла не должен печататься в выходном файле, но такой узел может иметь печатаемых сыновей, в таком случае они печатаются, и для их размещения может использоваться синтаксическое правило родителя. Привязка узлов с таким креатором игнорируется, даже если она корректна (например, узел является копией узла из текста).

BRE — узел создан в результате преобразующих проходов (функциональность для преобразования программ в Modernization Workbench называется Business Rule Extraction). Может ставиться на вновь созданные узлы или на копии существующих. Такие узлы должны печататься по дереву, используя синтаксические правила для размещения.

Прочие — ещё несколько креаторов, обозначающих, что узел появился в результате специализированных синтезирующих проходов. Могут применяться синтезирующими проходами для встроенных (Embedded) языков — SQL, CICS, и т.д. Генерация должна игнорировать такие узлы как PARSER_AUX. Узлы встроенных языков, текст которых присутствовал в исходной программе, помечены креатором PARSER.

3.4 Возможные виды генерации

Как уже упоминалось выше, используются два способа формирования текста для данного узла: чтение его из входного файла и вставка без изменений в выходной файл (далее будем говорить “генерация по привязке”), и построение текста узла по информации, сохранённой в дереве, с использованием правил синтаксиса данного языка программирования (генерация по дереву). Первый способ предпочтительней, поскольку гарантирует сохранение текста программы в исходном виде. Поскольку для задачи реинжиниринга именно сохранение программы в возможно более неизменном виде является приоритетом, основной целью алгоритма будет максимизация объёма текста, генерируемого по привязке. Качество генерации по дереву зависит в основном от сохранённой в дереве информации, поэтому в данной работе этому аспекту уделяется мало внимания. Единственным важным требованием к генерации по дереву является то, что даже в случае, если в дереве вообще нет информации о привязке, генерация должна быть способна сгенерировать корректную и правильно (с точки зрения ряда эвристических критериев) отформатированную программу.

3.5 Трудности

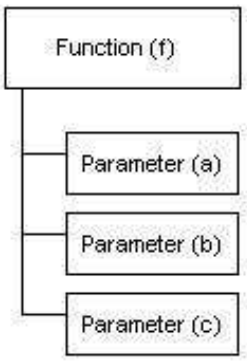
К несчастью, задача не имеет тривиального решения, сводящегося к тому, что все узлы, изменённые в процессе преобразований, будут печататься по дереву, а остальные — по привязке. Дело в том, что изменение текстового представления одного узла может привести к значительному изменению в расположении фрагментов текста программы. Рассмотрим ряд простых примеров, поясняющих некоторые из трудностей, которые заставляют делать изменения, которые могут показаться пользователю ненужными:

1. Простейший случай: удаление сына. Положим, в исходной программе имелся вызов функции:

   ```
   f(a, b, c)
   ```

   Положим, что данный текст представляется в дереве, как показано на рис. 1. Положим, один из параметров, скажем b, был удалён в

   Рис. 1:

   результате преобразований. В таком случае применение наивного подхода привело бы к

   ```
   f(a, , c)
   ```

   что нарушило бы синтаксическую корректность программы. Так как дерево не хранит информации о том, какой именно фрагмент текста узла родителя надо изменить при удалении того или иного сына, придётся перегенерировать весь текст родителя по дереву.

2. Теперь представим себе, что сыновья данного узла могут встречаться в тексте программы в произвольном порядке. Пример из Кобола:

   ```
   01 A PIC X OCCURS 10 и 01 A OCCURS 10 PIC X.
   ```

   Обе конструкции являются корректным объявлением массива символов из 10 элементов, порядок выражений OCCURS 10 и PIC X несущественен для семантики программы, поэтому никак не представлен в дереве. Положим, текстовое представление узла поменялось (переименовали переменную), тогда мы должны будем сгенерировать узел по дереву, а сыновей — по привязке. В каком порядке они будут в выходном файле? Как правильно включить текст сыновей в генерируемый по дереву текст родителя? Если исходная конструкция имеет привязку, показанную на рис. 2, т.е. PIC и

   Рис. 2:

   OCCURS являются частью текста родителя, при наивном подходе результирующий текст может стать

   ```
   01 A PIC 10 OCCURS X
   ```

   что опять-таки нарушит корректность программы.

3. Совместим примеры 1 и 2 и представим себе, что узлом, у которого сыновья могут идти в произвольном порядке и один из них удалён, является какая-нибудь большая структурная часть программы, например, секция или параграф в языке COBOL. В таком случае одно локальное изменение может привести к тому, что значительные фрагменты текста поменяются местами, множество текстовых промежутков сгенерируются по дереву, т.е. потеряют исходное форматирование.

3.6 Ограничения на дерево

Из сказанного выше следует, что задача генерации с использованием привязки требует наложения некоторых ограничений на синтаксическое дерево. Тем не менее, поскольку алгоритм должен использоваться в уже существующей системе, в которой, к тому же, генерация преобразованных программ не является единственной функциональностью, алгоритм не только не должен накладывать новые существенные ограничения, но и по возможности ослабить старые. Синтаксические анализаторы разрабатывались и сопровождались с учётом требований использовавшегося ранее алгоритма. Поэтому было известно, что информации в синтаксическом дереве достаточно для генерации по дереву и, в большинстве случаев, достаточно для генерации по привязке. В текущей реализации для входного дерева выполнены следующие ограничения:

- Информации в дереве достаточно для генерации текстового представления любого узла дерева (и его сыновей) без использования привязки.
- Привязка родителя (если она задана) включает в себя привязки всех сыновей.
- Привязка узла включает в себя только текст самого узла и его сыновей. Она также может включать в себя и другой текст, если он не принадлежит узлу в дереве (например, комментарий или директива препроцессора).
- Весь текст узла содержится внутри текстового промежутка, задаваемого привязкой узла.

Для предлагаемого алгоритма часть этих ограничений несущественна. Далее, в процессе описания алгоритма будет показано, как можно снять определённые ограничения, и будут предложены некоторые дополнительные соглашения, которые помогли бы упростить алгоритм генерации или повысить качество генерируемых программ.

3.7 Цели

Итак, основные цели разработки алгоритма генерации таковы:

1. Сгенерированный текст программы должен содержать возможно меньше промежутков, сгенерированных по дереву.

2. Алгоритм должен накладывать возможно меньше ограничений на входное синтаксическое дерево и на допустимые преобразования.

3. Алгоритм должен содержать как можно меньше недостатков, не позволяющих сгенерировать корректную программу в сложных случаях.

4. Промежутки, сгенерированные по дереву, должны выглядеть как можно более похоже (неформально) на текст исходной программы.

5. Реализация алгоритма должна позволять быстро поддерживать новые языки программирования.

4 Обзор алгоритма, применявшегося ранее

4.1 Описание алгоритма

4.1.1 Терминология

Дерево — абстрактное синтаксическое дерево программы.

Узел — узел дерева.

Привязка — позиция в тексте (номер файла, строка и колонка).

Промежуток — часть текста исходного файла, характеризуется точками начала и конца, а также включены или нет в него конец/начало.

Копибук (от англ. Copybook) - файл, включаемый в главный файл исходной программы перед или во время компиляции. **COPY** далее будет называться любая инструкция, включающая такой файл (В языке COBOL это может быть COPY, в RPG - /COPY, в C/C++ - #include). Так же мы будем называть конструкции, неявно включающие в код программы какие-либо файлы, например, использование внешне описанных структур данных, подключение описания базы данных, использование описания экранной формы и т.д.

Трансформирующий проход - этап работы системы, так или иначе преобразующий синтаксическое дерево.

Переключение файлов - смена текущего генерируемого выходного файла, например, при начале генерации копибука.

Разбитый копибук - копибук, узлы из которого оказались в процессе преобразований дерева разделены узлами из другого файла (например, разнесены по разным частям программы). Разбитый копибук не может быть сгенерирован как единое целое, поэтому в целевой программе создаётся новый копибук для каждой части исходного копибука.

Регенерация или **перегенерация** копибука — генерация нового копибука по изменённому и исправление оператора его включения. В случае, если копибук не изменён, новый копибук не генерируется, а сохраняется COPY на старый копибук.

4.1.2 Входные данные

На вход алгоритму генерации поступает дерево программы, преобразованное трансформирующими проходами, таблица файлов с информацией об иерархии включения копибуков и положении соответствующих COPY в тексте программы, и исходный текст. Для дерева выполнены все соглашения из 3.6, а также соглашения об атрибутах и креаторах. Возможные настройки генерации включают печать/не печать удалённых узлов как комментариев, сохранение исходных копибуков или генерация новых, если были изменения, пометка модифицированных/удалённых узлов специальными комментариями, некоторые другие.

4.1.3 Алгоритм

Алгоритм генерации работает в три этапа:

1. Анализ дерева.
2. Препроцессирование дерева.
3. Генерация текста.

Алгоритм обхода дерева по синтаксическим правилам (алг. A). На этапах 1 и 3 обход дерева выполняется одинаково, различаются лишь действия, выполняемые во время обработки промежутков:

1. Дерево обходится в порядке, описанном в синтаксических правилах.
2. При посещении каждого узла обрабатываются промежутки и сыновья узла, следующим образом:
   (a) Обрабатывается промежуток от начала узла до начала первого сына (если он есть).
   (b) Обрабатывается сын.
   (c) Обрабатывается промежуток между привязкой конца первого сына и привязкой начала второго.
   (d) и т.д. для всех сыновей.
   (e) Обрабатывается промежуток от конца последнего сына до конца узла.

3. Если один из сыновей не имеет привязки, промежутки до и после такого сына считаются некорректными. В этом случае работает режим обработки по дереву для узла. Другие сыновья, тем не менее, могут обрабатываться по привязке.

Анализ. Анализ дерева собирает информацию об изменённых и удалённых промежутках, которая впоследствии может быть использована для подсчёта статистики изменений и, что более важно для генерации текста, для определения копибуков, которые были изменены и нуждаются в перегенерации.

1. Анализ начинается с прохода по таблице файлов — так называемой симуляции препроцессора. Цель — разбить текст программы на разделы так, чтобы каждый раздел был непрерывным куском текста из одного файла. Потом эта информация будет использоваться для разбиения многофайловых промежутков на однофайловые.
   (a) Создаётся раздел от начала файла до первого COPY (привязка COPY указана в таблице файлов).
   (b) Анализируется файл, подключаемый COPY (с использованием этого же алгоритма).
   (c) Создаётся раздел от первого COPY до второго.
   (d) И так далее для всех COPY.
   (e) Последний раздел - промежуток от последнего COPY до конца файла.
2. Основная работа происходит во время обхода дерева, где обработка подразумевает анализ промежутка по алгоритму A.
   (a) Если промежуток относится к модифицированному узлу (т.е. модифицирован узел, при анализе которого был порождён этот промежуток), промежуток считается модифицированным, если к удалённому — удалённым.
   (b) Также происходит эмуляция печати текста в выходной файл, целью которой является отслеживание переключения файлов.
   (c) Если во время анализа файл был открыт более одного раза, он помечается как разбитый.
   (d) Если все промежутки из копибука удалены, копибук считается удалённым. Если в копибуке есть изменённые промежутки,

при этом не все удалены, он считается изменённым. В случае, если установлена опция не генерировать новые копибуки, этот флаг сбрасывается.

Препроцессор Цель препроцессора — провести подготовку дерева к генерации — разметить узлы методами печати (см. далее), удалить лишние узлы, провести некоторые преобразования дерева.

1. Из дерева удаляются вспомогательные узлы, созданные трансформирующими проходами или синтаксическим анализатором.
2. Узлы, про которые заведомо известно, что у них неправильная привязка (или привязка, непригодная для генерации), помечаются как печатаемые по дереву, или их привязка сбрасывается.
3. Узлы помечаются как генерируемые по привязке, если сами они и их сыновья не требуют генерации по дереву.
4. Все остальные узлы никак не помечаются.
5. Происходит перемещение комментариев (о том, как представляются комментарии в дереве и зачем выполняется перемещение, будет пояснено отдельно).

Генератор текста Генератор текста работает следующим образом: обходит дерево, используя стандартный алгоритм обхода по синтаксическим правилам (алгоритм A), где обработка ведётся по следующим правилам:

1. Если узел может быть распечатан целиком по привязке (должен быть помечен препроцессором), он печатается сразу, одним промежутком, соответствующим привязке узла. Синтаксическое правило для данного узла не используется, соответственно, обход сыновей не происходит.
   (a) Печать промежутка происходит следующим образом: сначала промежуток разбивается на однофайловые промежутки, используя информацию, собранную анализатором на шаге 1.
   (b) Для каждой получившейся части промежутка происходит проверка на необходимость переключения файла (проверяется, что привязка начала промежутка из текущего файла).
   (c) Если файл надо переключить, генерируется соответствующий COPY и файл переключается.

(d) Генератор посимвольно читает текст из исходного файла в заданном промежутке и вставляет его в выходной файл.

2. Если узел не может быть распечатан целиком по привязке, вызывается процедура печати узла согласно синтаксису. Синтаксис узла жёстко закодирован.

   (a) Если узел должен быть распечатан по дереву, печатается текст, определённый в синтаксических правилах для данного промежутка (промежутки генерируются в последовательности, определённой алгоритмом A). Для печати отступа (если она необходима) используется информация о привязке узла (текстовое представление узла в этом случае, как правило, изменено, но сохранить его позицию в тексте программы мы можем).

   (b) Если узел не должен печататься по дереву, полученный в процессе работы алгоритма A промежуток печатается как описано в пунктах 1a - 1d.

   (c) Во всех случаях печатается комментарий следующего сына, если он есть.

3. Если узел удалён, то его печать происходит либо с включённым режимом комментирования, либо не происходит вообще (в зависимости от опции).

Замечания Комментарии не имеют своих узлов в дереве, комментарий — поле узла. Считается, что комментарий в исходном коде относится к тому узлу, который расположен за ним (текстуально или в порядке обхода дерева, в зависимости от реализации). Тем не менее, комментарии имеют свою привязку.

4.2 Недостатки алгоритма

Порядок узлов. Порядок обхода узлов дерева по синтаксическим правилам во время генерации должен соответствовать порядку соответствующих конструкций в тексте программы. Иначе может случиться так, что второй сын попадёт в промежуток между началом узла и первым сыном. В таком случае он распечатается дважды. Синтаксический анализатор не всегда может обеспечить такое соответствие, например, некоторые диалекты языка COBOL позволяют записывать части синтаксиса в произвольном порядке. Для иллюстраци используем приведённый выше пример с OCCURS:

```
01 A OCCURS 10 PIC X.
```

Положим, в дереве фиксирован порядок

```
01 A PIC X OCCURS 10.
```

Тогда алгоритм распечатает сначала промежуток от начала узла до первого сына (X), затем первый сын, затем от первого сына до второго (10), промежуток получится некорректным и поэтому не будет распечатан), затем второй сын, затем от второго сына до конца узла. Получим

```
01 A OCCURS 10 PIC X 10 PIC X.
```

Проблема решается использованием генерации по дереву объемлющих узлов, но, очевидно, это приводит к снижению качества результатов.

Печать элементов, не являющихся частью синтаксической структуры программы. Примеры таких элементов текста — комментарии и директивы препроцессора. Поскольку они не являются частью синтаксиса, они не могут быть посещены при обходе дерева по синтаксическим правилам. Такие элементы могут быть распечатаны, если промежуток, в который они попали, будет печататься по привязке, т.е., фактически, случайно и вне зависимости от самих директив или комментариев. Для директив препроцессора ситуация ещё хуже — синтаксическое дерево строится по препроцессированной

программе, поэтому части текста, удалённые препроцессором, могут быть сгенерированы по привязке. Для комментариев решение существует — комментарии подвешиваются в дерево к узлам, которые они комментируют. В таком случае комментарий можно распечатать перед печатью узла. Но существует опасность, что комментарий сгенерируется дважды — по привязке как текст промежутка между двумя узлами и как комментарий изменённого узла. К тому же, это создаёт дополнительные проблемы при переключении файлов — могут сгенерироваться копибуки, состоящие только из комментариев. Для обхода таких проблем используются перемещения комментариев по дереву перед генерацией, тем не менее, некоторые комментарии могут быть потеряны при переключении файлов. Директивы препроцессора можно подвешивать в дерево как комментарии, но это накладывает ограничения на перемещение узлов. В некоторых случаях директивы можно подвесить в дерево как узлы, например, директивы управления режимами компиляции /FREE и /END-FREE в языке RPG. Они могут нарушать структуру вложенности операторов, что помешает, например, удалению узлов — даже если родитель удалён, /FREE всё равно должна быть распечатана. Это порождает проблему: нельзя удалить пару /FREE — /END-FREE, даже если она действительно не нужна. В общем случае такое решение не проходит, т.к. директивы могут быть везде, в т.ч. разрывать синтаксис узла.

Разрывы привязки. Имеем какой-то сложный узел, между его сыновьями в тексте стоят другие узлы, которые встретятся в другой части дерева. Например, конструкция из языка RPG:

```
MONITOR <Body> <OnError> ENDMON.
```

ENDMON является частью Body, но между ним и Body находятся узлы из синтаксической конструкции OnError. Получается, что когда Body генерируется по привязке, все конструкции из OnError печатаются по привязке промежутка между последним узлом из Body и ENDMON. Потом генерация доходит до OnError и печатает его ещё раз. В таком случае необходимо сгенерировать узел Body по дереву, что может привести к потере форматирования довольно большого фрагмента программы. Ещё один пример из языка RPG:

```
D Variable1         S
```

```
D Constant1         C                   1
D Variable2         S
D Constant2         C                   2
```

В этом примере объявляются две константы и две переменные. В RPG объявления констант и переменных могут идти в произвольном порядке. Все переменные, описанные в программе, являются сыновьями одного узла, все константы — другого. Их привязки могут перекрываться, что приведёт к печати лишнего текста по привязке промежутка между узлами. В приведённом выше примере определение Constant1 и Variable2 может быть распечатано дважды. Приходится печатать объемлющие узлы по дереву, что приведёт к потере форматирования (а также некоторых комментариев и директив препроцессора) в промежутках между определениями.

Переключение файлов. 1. Потеря информации о необходимости переключения файлов. Пример из языка COBOL — определение массива WS10-CHARGE-KEY-REC, число элементов массива указано в копибуке ara.cpy. Вторая точка заканчивает определение, в копибуке ara2.lb находятся другие определения.

```
01 WS10-CHARGE-KEY-REC   OCCURS COPY  'ara.cpy'..
  COPY 'ara2.lb'.
```

Предположим, что имя декларации поменялось и узел генерируется по дереву. Во время генерации значения OCCURS произойдёт переключение на выходной файл для ara.cpy, а точка будет сгенерирована по синтаксическому правилу, без использования информации о привязке. Файл не будет переключен. Если копибук не перегенерируется (такое может произойти, если пользователь решит оставить старые копибуки), точка будет потеряна. Сгенерированная программа не будет компилироваться.

2. Если у узла сброшена привязка, он будет генерироваться в текущий файл, что может привести к некорректности программы, в случае, если текущий файл не будет сгенерирован.

3. Могут создаваться копибуки из одних комментариев, может происходить разбиение копибуков из-за комментариев.

4. Между узлами из одного копибука в результате преобразований в дерево не должны попадать узлы из других файлов, иначе копибук будет разбит без необходимости.

5 Описание предлагаемого решения

5.1 Идеи, заложенные в основу алгоритма

- Желательно, чтобы новый алгоритм не зависел от порядка узлов в дереве, имеющейся в привязке информации должно хватать для размещения текстов узлов в том порядке, в котором они были в исходном файле.

- В таком случае можно обойтись без требования вложенности привязок (3.6). Если привязки известны, мы можем расположить тексты узлов правильно, даже если узлы не находятся ни в каком иерархическом отношении.

- Такой подход требует сортировки текстовых представлений узлов, что сложнее по времени, чем обход дерева, но позволяет снять излишние ограничения. Сортировать узлы потребуется, если мы хотим избежать хранения информации о порядке сыновей узла в дереве, но если мы будем сортировать все узлы в файле, мы сможем избавиться от ограничения на непрерывность привязки (3.6), что позволит нам естественным образом генерировать директивы препроцессора и комментарии — весьма важная для корректности результата цель.

- Поскольку генерация, как правило, выполняется после преобразующих проходов, которые имеют заведомо большую временную сложность, выбор между временем работы и точностью был сделан в пользу точности.

- Один узел в дереве может иметь несколько текстовых интервалов, для того, чтобы их сортировать, надо уметь их находить. Здесь желательна помощь синтаксических анализаторов, но уже имеющейся в дереве информации вполне достаточно: можно искать интервалы, обходя список привязок и анализируя уровень вложенности узлов. Привязка начала узла аналогична открывающейся скобке, привязка конца узла — закрывающейся. При таком подходе возникают три проблемы:

    1. Перекрывающиеся привязки:

       `Текст1Оператора1 Текст1Оператора2 Текст2Оператора1 Текст2Оператора2`

Используя аналогию со скобками, получим

```
([)]
```

В этом случае определение, к какому узлу какой текст относится, невозможно.

2. Не должно быть привязок, начинающихся в генерируемом файле и заканчивающихся в копибуке, на который нет оператора включения. Иначе конец привязки не удастся сравнить с остальными привязками (он будет в отдельном файле).
3. Не должно быть двух узлов, не связанных отношением предок-потомок, которые привязаны к одному и тому же текстовому интервалу. Иначе мы не сможем определить, кому из них принадлежит этот интервал.

Можно потребовать от входного дерева соблюдения этих ограничений, поскольку большинство из них требовалось и для старого алгоритма. Новым является только ограничение 3, поскольку мы хотим использовать привязку не только для генерации текста, но и для его размещения в выходном файле.

- В случае необходимости печати по дереву установить соответствие между текстом узла в исходном файле и текстом, сгенерированным по синтаксическому правилу невозможно, поскольку у узла может быть много текстовых промежутков, расположение которых никак не связано с синтаксическим правилом. Простой пример:

```
 01 SomeVariable
* Comment
 PIC X.
```

Положим, переменная переименована. В исходной программе она представлена двумя текстовыми промежутками (текст разбит комментарием), тогда как синтаксическое правило комментарий не учитывает. Поэтому надо генерировать изменённый узел по дереву, размещая непривязанные интервалы в порядке, определённом в синтаксических правилах. Привязка узла может быть учтена при размещении всего текста в выходном файле.

- Печать сыновей изменённого узла должна учитывать возможно изменившийся порядок его синтаксических конструкций. Пример:

```
01 SomeVariable PIC X VALUE 'A' OCCURS 100.
```

  Положим, без информации о привязке поддерево распечаталось бы по синтаксическим правилам так:

```
01 SomeVariable PIC X OCCURS 100 VALUE 'A'.
```

  В таком случае использовать привязку для размещения сыновей нельзя, поскольку

  1. Текст родителя будет распечатан без привязки, поэтому привязка сыновей не сможет подсказать, в каком месте внутри текста родителя размещены сыновья.
  2. Сыновья могут оказаться не в том порядке.

  В случае варианта 2 можно считать сыновья перемещёнными, если такой опасности нет (не важно, как расположены сыновья относительно текста родителя), можно печатать изменённый текст родителя вокруг сыновей с известной привязкой.

- То же касается новых узлов.

- Новые сыновья не дают возможности печатать родителя по привязке, поскольку невозможно определить, куда встраивать их синтаксис. Печатаются как модифицированные, со всеми ограничениями из предыдущих пунктов.

- Текст родителей, печатаемый по дереву, который должен быть расположен между сыновьями, должен печататься после сына, после которого он идёт в синтаксическом правиле. Так можно гарантировать, что все сыновья будут разделены текстом родителя (в таком случае порядок сыновей не важен, значит, не важно и какой именно текст будет распечатан после данного сына). Последний текстовый промежуток должен быть распечатан после всего узла, возможно, поменявшись местами с промежутком-разделителем.

5.2 Основные понятия

Точка привязки — точка в исходном файле, имеет координаты и два списка узлов: узлы, привязка которых начинается, и узлы, привязка которых заканчивается в данной точке. Такие узлы называются **родителями** точки привязки.

Интервал — объект, содержащий правило генерации фрагмента выходного текста. Над множеством интервалов определено отношение порядка, определяемое положением соответствующего им текста в генерируемом файле. В случае, если интервал имеет свою привязку (для неизменённых или изменённых узлов — привязка интервала соответствует привязке соответствующего промежутка в тексте), он добавляется в выходной файл в порядке, определённом привязкой. Если интервал не имеет привязки, его положение определяется привязкой интервала, добавленного перед ним. Т.е, для того, чтобы сравнить два интервала, не имеющих собственной привязки, нужно найти ближайшие интервалы перед ними, которые имеют привязку, и сравнить их. Если это один и тот же интервал, меньшим считается тот интервал без привязки, который был добавлен раньше.

Псевдоинтервал — интервал, используемый не для генерации текста, а для разметки выходного файла в случае, если интервалов недостаточно, чтобы сохранить всю информацию о привязке, имеющуюся в узле. Используется, например, чтобы указать точки начала/конца изменённых узлов (поскольку такая информация может быть потеряна из-за текстовых представлений сыновей, идущих до/после текста узла в исходном файле), а также для сохранения исходного положения перемещённого промежутка.

Текстовый копибук — копибук, оператор включения которого сводится к копированию текста копибука в текст программы на месте оператора включения. **Нетекстовый копибук** — копибук, оператор включения которого сводится к добавлению новых узлов непосредственно в синтаксическое дерево программы. Основным различием между этими видами копибуков для генерации является отсутствие во втором случае оператора COPY и возможности определить, какая точка привязки является “пограничной” между двумя файлами, т.е. если есть промежуток, начинающийся в одном файле и заканчивающийся в другом, в первом случае можно однозначно разбить этот промежуток на два однофайловых, во втором — нельзя.

Генерация нетекстовых копибуков, хотя и возможна, заключает в себе определённые трудности (связанные с заполнением промежутков между образами узлов), кроме того, обычно требует сложной модификации текста программы при исправлении оператора включения, поэтому в данной реализации не поддерживается. Например, в языке RPG примером нетекстового копибука является внешнее определение структуры данных. Имя переменной, описываемой такой структурой, совпадает с именем подключаемого файла, таким образом перегенерация файла требует переименования переменной во всей программе.

Значимый текст — текст, который синтаксический анализатор считает частью какой-либо конструкции. Например, пробелы и переводы строк (не разделяющие) значимым текстом обычно не являются (от их наличия или отсутствия корректность программы не зависит). Вместе с тем, комментарий является значимым текстом, поскольку имеет свой образ в дереве. Для языка COBOL интересным примером являются комментарии после 72 колонки, поскольку, хотя они не имеют образа в дереве и не анализируются синтаксически, они могут привести к некорректной программе, если будут смещены. Таким образом, текст комментария, распечатанный после 72 колонки, не является значимым, тогда как тот же текст, сдвинутый в колонки 6 - 72 будет значимым.

5.3 Шаги алгоритма

1. Препроцессор

   (a) Удаление лишних узлов из дерева. Лишними на данный момент считаются узлы, созданные BRE с атрибутом Deleted. Узлы не удаляются физически из дерева, а помечаются креатором PARSER_AUX, что скрывает их от генерации.

   (b) Рекурсивное распространение атрибута Deleted.

   (c) Исправление известных ошибок дерева: сброс привязки у тех узлов, у которых она заведомо неправильна (или может быть неправильной), установка PARSER_AUX на те узлы, которые не должны генерироваться. Замечание: сбрасывать привязку можно только тем узлам, которые не имеют своего представления в тексте программы (кроме незначимого текста). В случае, если у узла есть значимый текст и может быть неправильная

привязка (когда значимый текст оказывается вне привязки узла), алгоритм может сгенерировать неправильный выходной файл.

(d) На узел с перемещёнными или новыми сыновьями ставится атрибут Modified, если такие сыновья имеют своё текстовое представление.

(e) На сыновей изменённых узлов ставится атрибут Moved, если синтаксис узла зависит от порядка сыновей.

(f) Рекурсивное распространение атрибута Moved.

(g) Ставится PARSER_AUX на узлы из нетекстовых/системных копибуков и копибуков только для чтения. Это делается не только для экономии времени генерации, но и по тем соображениям, что узлы без привязки могут быть сгенерированы в копибук, никаких средств для определения, в какой файл генерировать узел нет, кроме эвристик. Если узел попадёт в негенерируемый копибук, он не будет сгенерирован вовсе, что может привести к неверифицируемости сгенерированной программы.

(h) Сбрасываются лишние атрибуты — если узел удалён и модифицирован, атрибут Modified сбрасывается. Сбрасываются привязки узлов, созданных трансформирующими проходами.

(i) Список может расширяться языкозависимыми шагами. (Например, для RPG на этом этапе производится анализ директив /FREE — /END-FREE и восстановление их структуры, если она нарушена трансформациями.)

2. Обход дерева в глубину

(a) Анализируются только узлы, созданные синтаксическим анализатором или трансформирующими проходами. Узлы с креатором PARSER_AUX игнорируются.

(b) Если у узла есть привязка, начало и конец узла оборачиваются в точки привязки, в них запоминается узел.

(c) Добавляем точки привязки в список: смотрим, есть ли уже точка привязки с такими координатами. Если да, добавляем узел в список родителей, если нет, создаём новую точку привязки и добавляем её в упорядоченный список. Узел добавляется в список родителей точки, соответствующий начинающимся/заканчивающимся в точке узлам, в порядке вложен-

ности промежутков узла (т.е, например, если узел добавляется в список узлов, начинающихся в данной точке, он будет добавлен в порядке убывания координат концов узлов). Если промежутки нельзя упорядочить по вложенности, это ошибка в представлении дерева (привязки перекрываются), возможна генерация некорректного результата.

(d) Такие списки точек привязки есть для каждого файла, заполняются независимо.

3. Добавляются точки привязки для узлов, хранящихся отдельно от дерева (на данный момент это привязки операторов COPY, хранящиеся в таблице файлов). Добавляются привязки, соответствующие началам и концам файлов.

4. Обход дерева по синтаксическим правилам

   (a) Перед узлом добавляются комментарии, генерируемые по дереву. Для того, чтобы гарантировать, что комментарии сгенерируются именно перед узлом, при генерации комментария используется информация о привязке узла.

   (b) Если текущий узел не изменён, все интервалы, относящиеся к нему, добавляются как генерируемые по привязке (все интервалы добавляются в описанном выше порядке, для поддержания упорядоченности используется специальный механизм, см. 5.4.2).

   (c) Если текущий узел новый (т.е. не имеет привязки), текст между началом узла и первым сыном, сгенерированый по синтаксическому правилу, вставляется как непривязанный интервал перед первым привязанным интервалом сына, если он есть, иначе после последнего известного привязанного интервала, анализируется первый сын, текст между первым и вторым сыном добавляется как непривязанный интервал после последнего интервала последнего проанализированного сына (если он известен), или после последнего интервала узла (в зависимости от того, должен ли текущий интервал печататься после всего сгенерированного текста узла, или после текущего сына, такая информация указывается в синтаксических правилах). Повторяем добавление текста для всех сыновей узла, текст между последним сыном и концом узла добавляется после последнего сгенерированного интервала узла.

(d) Если текущий узел удалён, все интервалы, относящиеся к нему, добавляются как генерируемые по привязке и помечаются как удалённые (метки на интервалах не влияют на их порядок, но требуются для того, чтобы помечать удалённые/изменённые интервалы в выходном файле).

(e) Если текущий узел изменён, добавляется псевдоинтервал с привязкой начала узла, после чего добавляется текст узла, сгенерированный по дереву, по той же схеме, что и на шаге 4c, новые интервалы помечаются как изменённые. Отметим, что псевдоинтервал влияет на порядок размещения интервалов так же, как и обычный текстовый интервал. После этого добавляется псевдоинтервал с привязкой конца узла.

(f) Если текущий узел перемещён, все интервалы, относящиеся к нему, помещаются после последнего интервала в порядке, в котором они встречались в тексте. Добавляем на их оригинальные позиции псевдоинтервалы с пометкой “перемещён”. Псевдоинтервалы с пометкой “перемещён” на размещение других интервалов не влияют, они служат только для хранения информации о наличии перемещённого текста.

(g) Если текущий узел помечен атрибутом BadlyConnected, в качестве текущего интервала, после которого добавляются непривязанные интервалы, устанавливается текущий интервал файла, в котором находился узел (аналог переключения файла из предыдущего алгоритма), для этого текущие интервалы хранятся для каждого файла, подробнее см. 5.4.2. Далее узел обрабатывается как новый.

5. Добавляются интервалы узлов, которые не были добавлены на предыдущем этапе (как правило те, которых не было в дереве), как генерируемые по привязке. Добавляются комментарии, генерируемые по привязке.

6. Интервалы, генерируемые по дереву и содержащие незначимый текст удаляются, если это возможно (если это интервалы между двумя интервалами, генерируемыми по привязке). Образовавшиеся промежутки будут заполнены в последующих проходах интервалами из исходного файла, что позволяет существенно повысить точность генерации. Замечание: на этом этапе расположение интервалов по колонкам ещё неизвестно, поэтому незначимый текст может быть принят за значимый. Однако, пробелы и переводы строк можно удалять, если текст из исходного файла, на который они будут

заменены, не может быть значимым ни в какой позиции (например, тоже пробелы и переводы строк. В случае, если текст содержит переводы строк, нам известна позиция в выходном файле всего текста за первым переводом строки, поэтому проверке подлежит только первая строка текста). Данная эвристика является аналогом свойства старого алгоритма, которое позволяло печатать текст между сыновьями по привязке, даже если узел содержал новые или удалённые сыновья. В силу отсутствия соглашений о порядке сыновей и вложенности привязки мы не можем воспроизвести это свойство в новом алгоритме, но практика показывает, что данной эвристики, как правило, вполне достаточно.

7. Обходятся получившиеся списки интервалов

   (a) Если в списке оказались перемещённые интервалы из другого файла, они выносятся в отдельный список. Список, соответствующий исходному файлу этих интервалов и новый список помечаются как изменённые, генерируются соответствующие операторы COPY.

   (b) Если список содержит только неизменённые интервалы, он удаляется из общего списка файлов. То же делается со списками, соответствующими нетекстовым копибукам и копибукам только для чтения.

   (c) Если копибук содержит только удалённые интервалы, все интервалы соответствующего ему оператора включения помечаются как удалённые.

   (d) Иначе модифицируется соответствующий этому списку оператор включения, так, чтобы он включал новый сгенерированный копибук.

8. Промежутки между интервалами заполняются следующим образом:

   (a) Удаляются псевдоинтервалы без флага “перемещён”.

   (b) Промежуток между интервалами, имеющими привязку (кроме помеченных, как перемещённые), добавляется как интервал, генерируемый по привязке.

   (c) Промежуток между интервалами, один из которых не имеет привязки или перемещён, не добавляется.

   (d) Промежуток перед псевдоинтервалом с флагом “перемещён” не добавляется. Сам интервал удаляется из списка.

9. Выполняется распространение пометки “удалён”. Для всех списков интервалов интервалы, добавленные на предыдущем шаге, расположенные перед удалёнными интервалами, помечаются как удалённые. Комментарии перед удалёнными интервалами также помечаются как удалённые.

10. Интервалы, добавленные на предыдущем шаге, удаляются или замещаются на перевод строки, если они содержат значимый текст (это позволяет частично компенсировать ошибки в привязке, а также избежать ошибок, когда на предыдущем шаге теряется информация, сохраняемая в псевдоинтервалах).

11. Выполняется разбиение на строки. Обходятся списки интервалов, храня информацию о текущей колонке и текст текущей строки. Для каждого интервала вызывается языкозависимая функция генерации — если интервал генерируется по привязке, его текст читается из файла и к нему применяется языкозависимое форматирование, которое учитывает текущую колонку и текст строки. Если интервал генерируется по дереву, вызывается языкозависимая функция, которая строит текст интервала по сохранённой информации из синтаксического правила с учётом текущей строки и колонки. Текст интервала добавляется к текущей строке, если интервал содержит перевод строки, по текущей строке формируется новый текстовый интервал (содержащий текст одной строки), и добавляется в выходной список. Если текущий интервал изменён, вся строка помечается как изменённая. Если текущий интервал удалён и не должен быть распечатан, строка помечается как имеющая удалённые интервалы (это может быть использовано для форматирования — печати отступа). Если текущий узел — комментарий, строка помечается как закомментированная. Если необходимо, создаётся новая строка. Если в закомментированную строку попадает незакомментированный интервал, создаётся новая строка. На выходе этого шага получаем последовательность текстовых интервалов — строк.

12. Печать результата: обходится список интервалов, сгенерированные на предыдущем шаге строки печатаются в файл, при этом они комментируются или помечаются как изменённые, в зависимости от пометок интервалов, могут выполняться дополнительные языкозависимые методы редактирования результирующего текста (например, вставка комментариев после 72 колонки в языке COBOL).

5.4 Вспомогательные алгоритмы

5.4.1 Алгоритм поиска соответствующих узлу промежутков

- Работаем на списках точек привязок, построенных на шаге 2. Ищем точку привязки, соответствующую началу узла.
- Идём по списку, пока не найдём точку конца узла, считаем уровень вложенности точек привязки (количество узлов, которые заканчиваются в этой точке и количество узлов, которые начинаются в этой точке).
  - Уделяем внимание узлам, начинающимся и заканчивающимся в одной точке.
  - Промежутки между точками привязки с соответствующим уровнем вложенности добавляем в список промежутков узла.
- Если встретилась точка привязки, соответствующая началу текстового копибука, переключаемся на список точек привязки, соответствующий копибуку. Добавляем, если необходимо, промежуток между началом узла и началом оператора COPY, а также от начала файла до первой точки привязки.
- Если встретили конец файла, переключаемся на объемлющий файл, в точку, соответствующую концу оператора COPY. Добавляем, если необходимо, промежуток между последней точкой привязки файла и концом файла.

Примечание: если синтаксические анализаторы будут предоставлять информацию о привязке узлов в виде набора промежутков, а не одного промежутка, включающего, возможно, тексты других узлов, можно было бы не использовать этот алгоритм, и не заниматься в генерации довольно трудоёмкой и по времени, и по памяти работой с точками привязки.

5.4.2 Механизм поддержания упорядоченности интервалов

Имеется стек последних (в смысле описанного порядка 5.2) интервалов для каждого узла. При начале генерации узла по синтаксическому правилу заводится новая запись на стеке, каждый добавляемый в выходной список интервал сравнивается с текущим содержимым вершины стека, если необходимо, интервал на вершине стека обновляется. По завершении анализа сына (при генерации нового промежутка узла)

его последний интервал оказывается на вершине стека и сравнивается с последним интервалом узла (который должен быть на стеке под ним). Интервал сына снимается со стека, интервал узла при необходимости обновляется. Если сын запросит последний интервал узла, для которого ещё не генерировались интервалы (если сын текстуально предшествует узлу), возвращается последний известный интервал. Последний интервал сына делается текущим, за ним будут добавляться новые непривязанные интервалы. Если требуется добавить интервал после всего текста узла, возвращается не текущий интервал, а вершина стека. Последние известные интервалы хранятся для каждого файла, переключение файла реализовано как смена текущего интервала на сохранённый для данного файла.

6 Результаты

- Удалось снять следующие ограничения с дерева:
  - *Привязка родителя (если она задана) включает в себя привязки всех сыновей.* Это позволяет при синтаксическом анализе привязывать узлы только к разобранному тексту, не заботясь о сыновьях. Кроме того, это позволяет добавлять привязанные узлы на этапе преобразований дерева без модификации родителя (если его синтаксис позволяет).
  - *Привязка узла включает в себя только текст самого узла и его сыновей.* Это весьма важное ограничение привязывало структуру дерева к структуре исходного текста. Узлы, могущие идти в произвольном порядке, из-за этого не могли быть разнесены по разным родителям. Это же ограничение мешало печати текста, который не соответствовал синтаксической структуре программы (например, директивы препроцессора).

- Удалось избавиться от следующих недостатков:

  Порядок узлов. Узлы упорядочиваются по привязке при генерации, что гарантирует сохранение порядка текста узлов исходного файла. Сравнение результатов алгоритмов см. в A.1.

  Печать элементов, не являющихся частью синтаксической структуры программы. Такие элементы могут храниться как набор текстовых промежутков отдельно от дерева (или как узлы в дереве, собранные в одном месте). Алгоритм сумеет расположить их в выходном файле по привязке. Если привязка неизвестна, такие узлы распечатаны не будут, поскольку нет возможности определить их положение в выходном файле. Исключением является генерация копибуков по схемам баз данных, в этом случае операторы COPY, необходимые для подключения всех описаний элементов данных, будут генерироваться подряд в начале файла. К счастью, в других случаях либо привязка несинтаксических элементов известна, либо они не должны быть сгенерированы (трансформирующие проходы работают только над синтаксическим деревом). Сравнение результатов алгоритмов см. в A.2.

  Разрывы привязки. Поскольку ограничение на дерево, требующее отсутствия в привязке узла текста других узлов, не свя-

занных отношением “предок-потомок” снято, данная проблема не проявляется.

Переключение файлов. 1. Потеря информации о необходимости переключения файлов является проблемой и для нового алгоритма, поскольку связана с недостатком узлов в дереве. Однако, потерять часть синтаксиса из-за негенерируемых копибуков новый алгоритм не может, поскольку добавление непривязанных интервалов идёт только в генерируемые файлы. То же касается проблемы сброшенной привязки и генерации копибуков.

2. Копибуки из одних комментариев не будут создаваться, поскольку разбиение текста на файлы осуществляется отдельным проходом, можно явно удалить пустые файлы, если по тем или иным причинам они появятся.
3. *Между узлами из одного копибука в результате преобразований в дерево не должны попадать узлы из других файлов* — при размещении привязанного текста его положение в дереве не учитывается, копибуки разбиваются только в том случае, когда имеются перемещённые узлы из копибука, разделённые узлами из другого файла, т.е. необходимость разбития копибука связана с явными преобразованиями дерева, а не особенностями его представления.

- Недостатки нового алгоритма:
  - Необходимо, чтобы дерево не содержало узлов, не связанных отношением “предок-потомок” и привязанных к одному промежутку, если они будут генерироваться по привязке. Это связано с тем, что привязка используется не только для печати, но и для размещения текста узла в выходном файле, в таком случае непонятно, какой из узлов должен быть расположен в указанном промежутке.
  - Если один из сыновей не имеет привязки, мы должны генерировать всего родителя по дереву. Старый алгоритм позволял генерировать по дереву только те части синтаксиса родителя, которые граничили с таким сыном. Связан этот недостаток с тем, что порядок сыновей заранее неизвестен, к тому же в промежутки между сыновьями могут попадать другие узлы.
  - Ошибки синтаксического анализатора, связанные с неверной

или отсутствующей привязкой, будут приводить к более тяжёлым последствиям, чем в старом алгоритме. Связано это, опять-таки с тем, что привязка используется для размещения текста. Кроме того, требуется при обходе посещать всех сыновей, поскольку их привязка может быть независима от привязки родителей (старый алгоритм требовал, чтобы привязка сыновей всегда являлась частью привязки родителя, поэтому, если узел неизменён, его можно сгенерировать по привязке целиком).

  - Алгоритм имеет большую временную сложность, чем старый алгоритм: $O(n * ln(n))$ вместо $O(n)$, где n — количество обрабатываемых узлов, поскольку требуется сортировка текстовых промежутков для всего файла.

- Особенности реализации:
  Реализация состоит из языконезависимой основной части, которая реализует механизмы размещения интервалов, и общие для всех языков проходы, такие как разбиение на строки или анализ копибуков. Основная часть работает в терминах интервалов и абстрактных узлов дерева. Обход дерева, по которому основная часть строит последовательность интервалов, осуществляют языкозависимые расширения, реализованные в виде отдельных библиотек, использующих основную часть. Они же занимаются генерацией текста по дереву с использованием синтаксических правил, специфичных для языка. Там же специфицируется, как именно основная часть должна разбивать текст на строки, как генерировать операторы включения копибуков, и прочие языковые особенности. Т.е., для того, чтобы поддержать новый язык, достаточно описать его синтаксис и правила обхода дерева, и определить набор методов, предоставляющих основной части информацию об языковых конструкциях, например, о представлении комментариев, о формате операторов включения и т.д.

7 Заключение

Предлагаемый алгоритм реализован и используется в системе Modernization Workbench для генерации текстов программ на языках COBOL и RPG. Кратко резюмируем описанные в части 6 результаты. Алгоритм позволяет:

- исправить ошибки генерации, которые не было возможности исправить в реализации старого алгоритма;
- генерировать элементы текста, которые не являются частью синтаксиса программы, например, директивы препроцессора или комментарии;
- большее количество текстовых промежутков генерировать по привязке, т.е. сгенерированный текст ближе к тексту исходной программы;

Кроме того, алгоритм обладает более мягкими требованиями к дереву, чем алгоритм, использовавшийся ранее, что позволило упростить синтаксические анализаторы и трансформирующие проходы, и дало возможность расширять их функциональность. Реализация алгоритма позволяет легко поддерживать новые языки программирования посредством определения для них синтаксических правил и описания некоторых языковых особенностей. Имеется возможность дополнять основной алгоритм языкозависимыми проходами.

A Сравнение результатов работы алгоритмов генерации

A.1 Произвольный порядок сыновей

Первый пример приводится на языке Cobol для пояснения проблемы с расхождением порядка сыновей некоторого узла программы в синтаксическом дереве и в исходном файле. В программе объявлены две переменные. Объявления 88 уровня — мёртвый код, который будет удалён, что приведёт к изменению текстов объявлений-родителей.

Исходная программа:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   test-random-order.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 a picture 9 value 1.
   88 key-a value 2.
01 b value 1 picture 9.
   88 key-b value 1.

PROCEDURE DIVISION.
    display a.
    display b.
    goback.
```

Dead Code Elimination, старый алгоритм:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DCE1.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 a PIC 9 VALUE 1.
01 b PIC 9 VALUE 1.


PROCEDURE DIVISION.
    display a.
    display b.
    goback.
```

Выражения PIC и VALUE сгенерированы большими буквами в фиксированном порядке — для них работала генерация по дереву.

Dead Code Elimination, новый алгоритм:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DCE2.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 a picture 9 value 1.
01 b value 1 picture 9.

PROCEDURE DIVISION.
    display a.
    display b.
    goback.
```

Несмотря на то, что переменные модифицированы, сыновья сгенерированы как в исходном файле.

A.2 Несинтаксические элементы

Второй пример приводится на языке RPG для пояснения проблемы с печатью элементов текста, не являющихся частью синтаксиса. Программа представляет собой объявление трёх элементов данных, один из которых "мёртвый". Две переменные, B и C объявляются во включаемом файле, если выполнено условие директивы компиляции /IF DEFINED. Пример получился несколько боле сложным, чем необходимо для демонстрации, поскольку для сравнения использовалась модификация старого алгоритма, вставляющая несинтаксические участки текста в явно генерируемые по привязке промежутки. Как мы увидим, такой подход имеет свои недостатки.

Исходная программа:

```
D A               S
 /DEFINE CompileBC
 /COPY BCDecls
C     A             DSPLY
C     B             DSPLY
```

Файл CompileBC.rpgcpy:

```
 /IF DEFINED(CompileBC)
D B               S
D C               S
 /ENDIF
```

Dead Code Elimination, старый алгоритм:

```
D A               S
 /DEFINE CompileBC
 /COPY BCDECLS-DCE1-0
C     A             DSPLY
C     B             DSPLY
```

Файл BCDECLS-DCE1-0.rpgcpy:

```
 /IF DEFINED(CompileBC)
D B               S
```

Директива /ENDIF в конце включаемого файла не сгенерировалась, так как промежуток, в котором она оказалась, не сгенерировался по привязке.

Dead Code Elimination, новый алгоритм:

```
D A               S
 /DEFINE CompileBC
 /COPY BCDECLS-DCE4-0
C     A             DSPLY
C     B             DSPLY
```

Файл BCDECLS-DCE2-0.rpgcpy:

```
 /IF DEFINED(CompileBC)
D B               S
 /ENDIF
```

Директива /ENDIF сгенерировалась, поскольку её текстовый промежуток был явно включён в выходной файл.

Список литературы

[1] *Казанский, Б.* Генерация программ на целевых языках в задачах реинжиниринга / Б. Казанский // Автоматизированный реинжиниринг программ. — Издательство С.-Петербургского университета, 2000.

[2] *Терехов, А.* История и архитектура проекта RescueWare / А. Терехов, Л. Элрих, А. Терехов // Автоматизированный реинжиниринг программ. — Издательство С.-Петербургского университета, 2000.

[3] *Akers, R. L.* Re-Engineering C++ Components Via Automatic Program Transformation / R. L. Akers, I. D. Baxter, M. Mehlich // Proceedings of the 12th Working Conference on Reverse Engineering. — 2005. — Pp. 13–22.

[4] *Carroll, J. J.* Unparsing RDF/XML: Tech. rep. / J. J. Carroll: HP Labs, 2001.

[5] IBM. — Cobol390 Language Reference, fifth edition, 2000. — September.

[6] *Cordy, J. R.* Source Transformation, Analysis and Generation in TXL / J. R. Cordy // PEPM'06, ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation. — Charleston, South Carolina: 2006. — January. — Pp. 1–11.

[7] IBM. — ILE RPG Reference, sixth edition, 2004. — May.

[8] *Ramsey, N.* Unparsing Expressions With Prefix and Postfix Operators / N. Ramsey // *Software Practice and Experience.* — 1998. — Vol. 28, no. 12. — Pp. 1327–1356.

[9] *Van Der Brand, M.* Generation of formatters for context-free languages / M. Van Der Brand, E. Visser // *ACM Transactions on Software Engineering and Methodology.* — 1996. — Vol. 5, no. 1. — Pp. 1–41.