* **Как дерево вычислений FormulaAST будет вычислять значение для операнда CELL?**
  + В FormulaAST::CellExpr::Evaluate мы будем получать значение ячейки из SheetInterface, используя sheet.GetCell(pos\_).
* **Как будет работать граф зависимостей?**
  + Граф зависимостей будет храниться в Sheet в виде std::unordered\_map<Position, std::unordered\_set<Position>> dependencies\_.
  + Ключ — позиция ячейки, значение — множество позиций ячеек, от которых зависит данная ячейка.
  + При изменении ячейки мы будем обновлять граф зависимостей и инвалидировать кэш зависимых ячеек.
* **Как реализовать кэш?**
  + Кэш будет храниться в Cell в виде Value cached\_value\_ и bool cache\_valid\_.
  + Кэш будет вычисляться при первом вызове Cell::GetValue и инвалидироваться при изменении зависимых ячеек.
* **В какой последовательности должны работать новые методы при изменении ячейки?**
  + Sheet::SetCell вызывает Cell::Set.
  + Cell::Set парсит формулу, получает список зависимостей.
  + Sheet::SetCell проверяет циклические зависимости.
  + Sheet::UpdateDependencies обновляет граф зависимостей.
  + Sheet::InvalidateCache инвалидирует кэш зависимых ячеек.
  + Cell::Set устанавливает новое значение ячейки.

**Вершины и ребра графа**

* **Вершины:** Ячейки таблицы (представленные Position).
* **Ребра:** Зависимости между ячейками.
  + Прямые связи (ячейка A зависит от ячейки B).
  + Обратные связи (ячейка B зависит от ячейки A).
* Два типа связей необходимы для эффективного обновления графа и инвалидации кэша.

Эти изменения позволят реализовать новые функциональные возможности и обеспечить корректную работу электронной таблицы.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Вопрос: Подумайте, откуда дерево FormulaAST будет брать значения операндов CELL?

Ответ: из переданного в Execute const SheetInterface& sheet.

Вопрос: Придумайте, как реализовать для CellExpr : Expr метод Evaluate(...). То есть «научите» его находить нужную ячейку с нужной формулой, зная лишь позицию ячейки. Если нужно, добавьте в класс поля и методы или поменяйте сигнатуры — кроме интерфейсов?

Ответ: **Получение ячейки:**

* Мы используем sheet.GetCell(pos\_) для получения ячейки по ее позиции.

**Обработка пустой ячейки:**

* Если ячейка пуста (cell == nullptr), мы возвращаем 0.0.

**Получение значения ячейки:**

* Мы используем cell->GetValue() для получения значения ячейки.

**Обработка различных типов значений:**

* Если значение ячейки — число (double), мы возвращаем его.
* Если значение ячейки — строка (std::string), мы пытаемся преобразовать ее в число с помощью std::stod. Если преобразование не удалось, мы выбрасываем ошибку FormulaError::Category::Value.
* Если значение ячейки — ошибка (FormulaError), мы пробрасываем эту ошибку.

**Проброс ошибок:**

* Если при вычислении значения ячейки возникла ошибка (FormulaError), мы пробрасываем эту ошибку, чтобы она могла быть обработана на верхнем уровне.

Вопрос: Обратите внимание на интерфейс FormulaInterface из файла formula.h. В нём метод Evaluate имеет новую сигнатуру, у метода есть аргумент — ссылка на таблицу. Подумайте, откуда он будет поступать, куда и как затем передаваться.

Ответ: из класса Cell, куда будет передан по ссылке.

**Откуда поступает SheetInterface& sheet?**

* Этот параметр поступает из контекста, в котором вычисляется формула. Обычно это происходит внутри класса Cell, когда вызывается метод GetValue().
* Когда ячейка содержит формулу, для ее вычисления необходимо знать значения других ячеек, на которые она ссылается. Эти значения хранятся в таблице, представленной интерфейсом SheetInterface.
* Таким образом, SheetInterface& sheet предоставляет формуле доступ к данным таблицы, необходимым для вычисления.

**Куда и как передается SheetInterface& sheet?**

* **Из Cell::GetValue() в Formula::Evaluate():**
  + Когда вызывается Cell::GetValue(), если ячейка содержит формулу, создается объект Formula.
  + Затем вызывается метод Evaluate() этого объекта, и в качестве аргумента передается ссылка на текущую таблицу (\*sheet\_).
  + Пример кода:

C++

CellInterface::Value Cell::FormulaImpl::GetValue(const SheetInterface& sheet) const override {

FormulaInterface::Value value = formula\_ptr\_->Evaluate(sheet);

// ...

}

* **Из Formula::Evaluate() в FormulaAST::Execute():**
  + Formula делегирует вызов Evaluate() объекту FormulaAST, передавая ему ту же ссылку на таблицу.
  + Пример кода:

C++

FormulaInterface::Value Formula::Evaluate(const SheetInterface& sheet) const override {

return formula\_ast\_.Execute(sheet);

}

* **Из FormulaAST::Execute() в ASTImpl::Expr::Evaluate():**
  + FormulaAST::Execute() рекурсивно обходит дерево формулы и вызывает метод Evaluate() для каждого узла.
  + На каждом шаге обхода ссылка на таблицу передается в качестве аргумента.
  + Пример кода:

C++

double FormulaAST::Execute(const SheetInterface& sheet) const {

return root\_expr\_->Evaluate(sheet);

}

* **В ASTImpl::CellExpr::Evaluate():**
  + Когда обход дерева доходит до узла CellExpr, метод Evaluate() этого узла использует ссылку на таблицу для получения значения ячейки по ее позиции.
  + Пример кода:

C++

double CellExpr::Evaluate(const SheetInterface& sheet) const {

const CellInterface\* cell = sheet.GetCell(pos\_);

// ...

}

**3. Роль параметра SheetInterface& sheet**

* Этот параметр обеспечивает доступ к данным таблицы на всех этапах вычисления формулы.
* Он позволяет формуле получать значения других ячеек, необходимые для ее вычисления.
* Он обеспечивает гибкость и расширяемость, позволяя использовать различные реализации таблицы, соответствующие интерфейсу SheetInterface.

Таким образом, ссылка на таблицу передается от ячейки к формуле и далее к узлам дерева формулы, обеспечивая доступ к данным таблицы на всех этапах вычисления.

**Вопрос: Как будет работать граф зависимостей?**

Как ячейка будет хранить информацию о своей принадлежности к графу? Какие поля и методы надо добавить и куда (в какой/какие классы)?

Ответ: в класс Sheet в виде std::unordered\_map<Position, std::unordered\_set<Position>, PositionHash> dependencies\_; а также в каждой ячейке Cell формульного типа в виде std::unordered\_set<Position, PositionHash> referenced\_cells\_;

Вопрос: В программе мы добавили в лексический анализатор следующее: когда пользователь задаёт текст формульной ячейки в методе Cell::Set(), наш лексический анализатор возвращает std::forward\_list<Position> cells\_ с индексами ячеек, входящих в формулу. Как, имея эту информацию, встроить новую ячейку в граф зависимостей?

Что будет, если до вызова метода Cell::Set() ячейка была пустой?

Что будет, если она не была пустой, а содержала формулу с другими зависимостями и была встроена в граф? Как изменение в ней повлияет на другие ячейки?

Ответ: **1. Встраивание новой ячейки в граф зависимостей**

1. **Получение зависимостей:** После парсинга формулы в Cell::Set() мы получаем std::forward\_list<Position> cells\_ с индексами ячеек, от которых зависит текущая ячейка.
2. **Обновление графа:** Мы передаем этот список в метод Sheet::UpdateDependencies(Position pos, const std::unordered\_set<Position>& new\_dependencies), который обновляет граф зависимостей.
   * В этом методе мы:
     + Добавляем в dependencies\_ записи о том, что текущая ячейка зависит от ячеек из cells\_.
     + Обновляем обратные зависимости: для каждой ячейки из cells\_ добавляем в ее список зависимостей текущую ячейку.

**2. Ячейка была пустой до вызова Cell::Set()**

* Если ячейка была пустой, то в графе зависимостей для нее не было никаких записей.
* При вызове Cell::Set() мы просто добавляем новые зависимости в граф, как описано выше.

**3. Ячейка не была пустой и содержала формулу с другими зависимостями**

* Если ячейка не была пустой, то в графе зависимостей для нее уже были записи.
* При вызове Cell::Set() мы должны:
  1. **Удалить старые зависимости:** Удалить из графа записи о зависимостях, которые были у ячейки до вызова Cell::Set().
  2. **Добавить новые зависимости:** Добавить в граф записи о новых зависимостях, полученных после парсинга новой формулы.
  3. **Обновить обратные зависимости:** Обновить списки зависимостей для ячеек, от которых зависела старая формула, и для ячеек, от которых зависит новая формула.
* **Влияние на другие ячейки:**
  1. Изменение зависимостей ячейки может повлиять на другие ячейки, которые зависят от нее.
  2. Мы должны инвалидировать кэш всех ячеек, которые зависят от измененной ячейки, чтобы они пересчитали свои значения.
  3. Это делается с помощью метода Sheet::InvalidateCache(Position pos), который рекурсивно инвалидирует кэш всех зависимых ячеек.

**4. Проверка циклических зависимостей**

* Перед добавлением новых зависимостей в граф мы должны проверить, не приведут ли они к циклическим зависимостям.
* Это делается с помощью метода Sheet::HasCircularDependencies(Position pos, const std::unordered\_set<Position>& dependencies).
* Этот метод использует алгоритм поиска в глубину (DFS) для проверки наличия цикла в графе.
* Если цикл найден, мы выбрасываем исключение CircularDependencyException.

**5. Пример кода (псевдокод)**

C++

void Sheet::UpdateDependencies(Position pos, const std::unordered\_set<Position>& new\_dependencies) {

// Удаление старых зависимостей

if (dependencies\_.count(pos)) {

for (const auto& dep : dependencies\_[pos]) {

// Удаление обратной зависимости

// ...

}

dependencies\_.erase(pos);

}

// Добавление новых зависимостей

dependencies\_[pos] = new\_dependencies;

for (const auto& dep : new\_dependencies) {

// Добавление обратной зависимости

// ...

}

// Инвалидация кэша

InvalidateCache(pos);

}

void Sheet::InvalidateCache(Position pos) {

// Инвалидация кэша для ячейки pos

// ...

// Рекурсивная инвалидация кэша для зависимых ячеек

for (const auto& dep : dependencies\_) {

if (dep.second.count(pos)) {

InvalidateCache(dep.first);

}

}

}

Эти шаги позволяют корректно обновлять граф зависимостей и обеспечивать целостность таблицы при изменении ячеек.

Вопрос: Как реализовать кэш?

В каком виде будет реализован кэш? Когда и кем он будет вычисляться?

Как будет использоваться кэш при вычислении формулы в одной конкретной ячейке? Как это повлияет на эффективность?

Если пользователь изменил некоторые ячейки в таблице, что произойдёт с кэшем во всех остальных ячейках таблицы?

Оцени эффективность (O-семантика) от введения кэша (до и после).

Ответ: **1. Вид кэша**

* Кэш будет реализован в классе Cell в виде поля Value cached\_value\_ и флага bool cache\_valid\_.
* cached\_value\_ будет хранить вычисленное значение ячейки (число или ошибку).
* cache\_valid\_ будет указывать, является ли кэшированное значение актуальным.

**2. Вычисление кэша**

* Кэш будет вычисляться при первом вызове метода Cell::GetValue(), если cache\_valid\_ равен false.
* Вычисление будет выполняться внутри Cell::GetValue(), после получения значения ячейки с помощью Impl::GetValue().
* После вычисления кэша cache\_valid\_ будет установлен в true.

**3. Использование кэша**

* При каждом вызове Cell::GetValue() будет проверяться значение cache\_valid\_.
* Если cache\_valid\_ равен true, будет возвращено значение из cached\_value\_ без повторного вычисления формулы.
* Это значительно повысит эффективность, так как повторные вызовы GetValue() для ячеек с формулами будут выполняться за O(1).

**4. Инвалидация кэша**

* Если пользователь изменил некоторые ячейки в таблице, кэш во всех ячейках, зависящих от измененных ячеек, должен быть инвалидирован.
* Это будет выполняться с помощью метода Sheet::InvalidateCache(Position pos), который рекурсивно инвалидирует кэш всех зависимых ячеек.
* После инвалидации кэша cache\_valid\_ будет установлен в false.

**5. Оценка эффективности**

* **До введения кэша:**
  + Каждый вызов Cell::GetValue() для ячейки с формулой требовал полного вычисления формулы, что могло занимать O(K) времени, где K — количество ячеек, от которых зависит формула.
* **После введения кэша:**
  + Первый вызов Cell::GetValue() для ячейки с формулой занимает O(K) времени.
  + Последующие вызовы Cell::GetValue() для той же ячейки, пока ее зависимости не изменились, занимают O(1) времени.
  + Изменение ячейки и инвалидация кэша зависимых ячеек занимает O(N) времени, где N — количество зависимых ячеек.

**6. Пример кода (псевдокод)**

C++

CellInterface::Value Cell::GetValue() const {

if (cache\_valid\_) {

return cached\_value\_;

}

cached\_value\_ = impl\_->GetValue(sheet\_);

cache\_valid\_ = true;

return cached\_value\_;

}

void Sheet::InvalidateCache(Position pos) {

Cell\* cell = GetCell(pos);

if (cell) {

cell->cache\_valid\_ = false;

}

for (const auto& dep : dependencies\_) {

if (dep.second.count(pos)) {

InvalidateCache(dep.first);

}

}

}

Введение кэша значительно повышает эффективность вычисления формул, особенно при многократном обращении к одним и тем же ячейкам.

Вопрос: **В какой последовательности должны работать новые методы при изменении ячейки?**

Когда пользователь меняет/задаёт ячейку, все вышеперечисленное должно происходить в том или ином виде.

Подумайте, что и как должно происходить при вызове методов Cell::Set(std::string text) и Sheet::SetCell(Position pos, std::string text).

Ответьте на вопрос, в какой последовательности ваши новые методы должны вызываться.

Ответ: **1. Sheet::SetCell(Position pos, std::string text)**

1. **Проверка позиции:** Проверяем, что pos корректна. Если нет, выбрасываем InvalidPositionException.
2. **Получение ячейки:** Получаем ячейку по позиции pos с помощью GetCell(pos).
3. **Вызов Cell::Set(text):** Вызываем метод Cell::Set(text) для изменения содержимого ячейки.
4. **Проверка циклических зависимостей:** Проверяем, не приведет ли изменение ячейки к циклическим зависимостям. Если приведет, выбрасываем CircularDependencyException и откатываем изменения.
5. **Обновление зависимостей:** Обновляем граф зависимостей с помощью UpdateDependencies(pos, new\_dependencies).
6. **Инвалидация кэша:** Инвалидируем кэш зависимых ячеек с помощью InvalidateCache(pos).

**2. Cell::Set(std::string text)**

1. **Парсинг формулы:** Если text начинается с =, парсим формулу с помощью ParseFormulaAST(text) и получаем список зависимостей new\_dependencies.
2. **Установка нового значения:** Создаем новый объект Impl (либо TextImpl, либо FormulaImpl) и устанавливаем его как impl\_.
3. **Обновление referenced\_cells\_:** Обновляем множество referenced\_cells\_ на основе new\_dependencies.
4. **Сброс кэша:** Устанавливаем cache\_valid\_ в false.

**3. Sheet::UpdateDependencies(Position pos, const std::unordered\_set<Position>& new\_dependencies)**

1. **Удаление старых зависимостей:** Если у ячейки pos были старые зависимости, удаляем их из графа.
2. **Добавление новых зависимостей:** Добавляем новые зависимости в граф.
3. **Обновление обратных зависимостей:** Обновляем списки зависимостей для ячеек, от которых зависит pos.

**4. Sheet::InvalidateCache(Position pos)**

1. **Инвалидация кэша:** Инвалидируем кэш ячейки pos.
2. **Рекурсивная инвалидация:** Рекурсивно инвалидируем кэш всех ячеек, которые зависят от pos.

**5. Sheet::HasCircularDependencies(Position pos, const std::unordered\_set<Position>& dependencies)**

1. **Поиск в глубину (DFS):** Используем DFS для проверки наличия цикла в графе зависимостей.
2. **Возврат результата:** Возвращаем true, если цикл найден, иначе false.

**6. Cell::GetValue()**

1. **Проверка кэша:** Если cache\_valid\_ равен true, возвращаем cached\_value\_.
2. **Вычисление значения:** Вычисляем значение ячейки с помощью impl\_->GetValue(sheet\_).
3. **Сохранение в кэш:** Сохраняем вычисленное значение в cached\_value\_ и устанавливаем cache\_valid\_ в true.
4. **Возврат значения:** Возвращаем cached\_value\_.

**7. ASTImpl::CellExpr::Evaluate(const SheetInterface& sheet)**

1. **Получение ячейки:** Получаем ячейку по позиции pos\_ с помощью sheet.GetCell(pos\_).
2. **Обработка пустой ячейки:** Если ячейка пуста, возвращаем 0.0.
3. **Получение значения ячейки:** Получаем значение ячейки с помощью cell->GetValue().
4. **Обработка различных типов значений:** Обрабатываем различные типы значений (число, строка, ошибка).
5. **Возврат значения:** Возвращаем вычисленное значение.

Эти шаги обеспечивают корректное обновление ячеек и графа зависимостей, а также эффективное вычисление формул с использованием кэша.

Диаграмма алгоритма для вычисления значения в ячейках:

graph TD

A[Начало: Вызов FormulaAST::Execute(sheet)] --> B{Корень дерева (root\_expr\_) существует?};

B -- Да --> C[root\_expr\_->Evaluate(sheet)];

B -- Нет --> D[Возврат 0.0];

C --> E{Тип узла?};

E -- Число --> F[Возврат числа];

E -- Оператор (+, -, \*, /) --> G[Рекурсивный вызов Evaluate для операндов];

G --> H[Выполнение операции и возврат результата];

E -- CellExpr (ссылка на ячейку) --> I[Получение ячейки из sheet.GetCell(pos\_)];

I --> J{Ячейка существует?};

J -- Да --> K[Получение значения ячейки cell->GetValue()];

J -- Нет --> L[Возврат 0.0];

K --> M{Тип значения?};

M -- Число --> N[Возврат числа];

M -- Строка --> O[Попытка преобразования в число];

O --> P{Преобразование удалось?};

P -- Да --> Q[Возврат числа];

P -- Нет --> R[Возврат FormulaError::Value];

M -- FormulaError --> S[Возврат ошибки];

F --> T[Конец];

H --> T;

Q --> T;

R --> T;

S --> T;

L --> T;

N --> T;

D --> T;

Конечно, давайте построим диаграмму алгоритма для обхода дерева формулы и вычисления значения в ячейках.

**Диаграмма алгоритма**

Фрагмент кода

graph TD

A[Начало: Вызов FormulaAST::Execute(sheet)] --> B{Корень дерева (root\_expr\_) существует?};

B -- Да --> C[root\_expr\_->Evaluate(sheet)];

B -- Нет --> D[Возврат 0.0];

C --> E{Тип узла?};

E -- Число --> F[Возврат числа];

E -- Оператор (+, -, \*, /) --> G[Рекурсивный вызов Evaluate для операндов];

G --> H[Выполнение операции и возврат результата];

E -- CellExpr (ссылка на ячейку) --> I[Получение ячейки из sheet.GetCell(pos\_)];

I --> J{Ячейка существует?};

J -- Да --> K[Получение значения ячейки cell->GetValue()];

J -- Нет --> L[Возврат 0.0];

K --> M{Тип значения?};

M -- Число --> N[Возврат числа];

M -- Строка --> O[Попытка преобразования в число];

O --> P{Преобразование удалось?};

P -- Да --> Q[Возврат числа];

P -- Нет --> R[Возврат FormulaError::Value];

M -- FormulaError --> S[Возврат ошибки];

F --> T[Конец];

H --> T;

Q --> T;

R --> T;

S --> T;

L --> T;

N --> T;

D --> T;

**Описание шагов**

1. **Начало:** Вызывается метод FormulaAST::Execute(sheet), где sheet — это ссылка на таблицу SheetInterface.
2. **Проверка корня:** Проверяется, существует ли корень дерева формулы (root\_expr\_).
   * Если корень не существует, возвращается значение 0.0.
   * Если корень существует, выполняется шаг 3.
3. **Вычисление узла:** Вызывается метод Evaluate(sheet) для корня дерева.
4. **Определение типа узла:** Определяется тип текущего узла дерева.
   * **Число:** Если узел представляет число, возвращается это число.
   * **Оператор:** Если узел представляет оператор (+, -, \*, /), рекурсивно вызывается метод Evaluate(sheet) для левого и правого операндов. Затем выполняется операция и возвращается результат.
   * **CellExpr:** Если узел представляет ссылку на ячейку (CellExpr), выполняются шаги 5-10.
5. **Получение ячейки:** Получается ячейка из таблицы с помощью sheet.GetCell(pos\_), где pos\_ — позиция ячейки.
6. **Проверка ячейки:** Проверяется, существует ли ячейка.
   * Если ячейка не существует, возвращается значение 0.0.
   * Если ячейка существует, выполняется шаг 7.
7. **Получение значения ячейки:** Получается значение ячейки с помощью cell->GetValue().
8. **Определение типа значения:** Определяется тип значения, полученного из ячейки.
   * **Число:** Если значение — число, оно возвращается.
   * **Строка:** Если значение — строка, выполняется попытка преобразования строки в число.
     + Если преобразование удалось, возвращается число.
     + Если преобразование не удалось, возвращается ошибка FormulaError::Value.
   * **FormulaError:** Если значение — ошибка, она возвращается.
9. **Конец:** Вычисление завершается, и результат возвращается.

Эта диаграмма наглядно показывает, как происходит обход дерева формулы и вычисление значения в ячейках.

Поиск циклических зависимостей с оптимизацией, обеспечивающей проход по каждой ячейке только один раз?

graph TD

A[Начало: HasCircularDependencies(pos, dependencies)] --> B[Создание множеств visited и recursionStack];

B --> C[Добавление pos в visited];

C --> D[Добавление pos в recursionStack];

D --> E{Для каждой dep из dependencies};

E -- Да --> F{dep не в visited?};

F -- Да --> G[Рекурсивный вызов HasCircularDependencies(dep, dependencies\_[dep])];

G --> H{Рекурсивный вызов вернул true?};

H -- Да --> I[Возврат true (цикл найден)];

H -- Нет --> J{dep в recursionStack?};

J -- Да --> I;

E -- Нет --> K[Удаление pos из recursionStack];

K --> L[Возврат false (цикл не найден)];

F -- Нет --> J;

J -- Нет --> E;

I --> M[Конец];

L --> M;

**Описание шагов**

1. **Начало:** Вызывается метод HasCircularDependencies(pos, dependencies), где pos — текущая ячейка, dependencies — множество зависимостей текущей ячейки.
2. **Инициализация:** Создаются два множества: visited для отслеживания посещенных ячеек и recursionStack для отслеживания ячеек в текущем пути рекурсии.
3. **Пометка текущей ячейки:** Текущая ячейка pos добавляется в множество visited и recursionStack.
4. **Обход зависимостей:** Для каждой зависимой ячейки dep из множества dependencies выполняются следующие шаги:
   * **Проверка посещения:** Если dep не была посещена (dep не в visited), выполняется шаг 5.
     + **Рекурсивный вызов:** Рекурсивно вызывается метод HasCircularDependencies(dep, dependencies\_[dep]).
     + **Проверка результата:** Если рекурсивный вызов вернул true, это означает, что цикл найден, и метод возвращает true.
   * **Проверка рекурсивного стека:** Если dep уже находится в recursionStack, это означает, что цикл найден, и метод возвращает true.
5. **Удаление из рекурсивного стека:** После обхода всех зависимостей текущая ячейка pos удаляется из recursionStack.
6. **Возврат результата:** Если цикл не был найден, метод возвращает false.
7. **Конец:** Выполнение метода завершается.

**Оптимизация**

* Использование множества visited позволяет избежать повторного посещения уже проверенных ячеек.
* Использование множества recursionStack позволяет быстро обнаруживать циклы в текущем пути рекурсии.

Эта диаграмма наглядно показывает, как происходит поиск циклических зависимостей с оптимизацией, обеспечивающей проход по каждой ячейке только один раз.

Покажи диаграмму алгоритма:

Инвалидация кэша и его оптимизация с проходом по ячейкам только один раз?

graph TD

A[Начало: InvalidateCache(pos)] --> B[Создание множества invalidated];

B --> C[Создание очереди queue];

C --> D[Добавление pos в queue и invalidated];

D --> E{queue не пуста?};

E -- Да --> F[Извлечение cell\_pos из queue];

F --> G[Получение ячейки cell = GetCell(cell\_pos)];

G --> H{cell существует?};

H -- Да --> I[cell->cache\_valid\_ = false];

I --> J{Для каждой dep\_pos в dependencies};

J -- Да --> K{dep\_pos не в invalidated?};

K -- Да --> L[Добавление dep\_pos в queue и invalidated];

K -- Нет --> J;

J -- Нет --> E;

H -- Нет --> E;

E -- Нет --> M[Конец];

**Описание шагов**

1. **Начало:** Вызывается метод InvalidateCache(pos), где pos — позиция ячейки, кэш которой нужно инвалидировать.
2. **Инициализация:** Создаются множество invalidated для отслеживания инвалидированных ячеек и очередь queue для обхода графа зависимостей в ширину.
3. **Добавление начальной ячейки:** Начальная ячейка pos добавляется в очередь queue и множество invalidated.
4. **Обход очереди:** Пока очередь queue не пуста, выполняются следующие шаги:
   * **Извлечение ячейки:** Извлекается позиция ячейки cell\_pos из начала очереди.
   * **Получение ячейки:** Получается ячейка cell из таблицы с помощью GetCell(cell\_pos).
   * **Проверка ячейки:** Проверяется, существует ли ячейка cell.
     + Если ячейка существует, выполняется шаг 5.
     + Если ячейка не существует, выполняется шаг 4.
5. **Инвалидация кэша:** Устанавливается флаг cell->cache\_valid\_ в false, инвалидируя кэш ячейки.
6. **Обход зависимостей:** Для каждой зависимой ячейки dep\_pos из множества зависимостей текущей ячейки dependencies выполняются следующие шаги:
   * **Проверка инвалидации:** Если dep\_pos еще не была инвалидирована (dep\_pos не в invalidated), выполняется шаг 7.
   * **Добавление в очередь:** dep\_pos добавляется в очередь queue и множество invalidated.
7. **Конец:** Выполнение метода завершается.

**Оптимизация**

* Использование множества invalidated позволяет избежать повторной инвалидации кэша для уже обработанных ячеек.
* Использование очереди queue обеспечивает обход графа зависимостей в ширину, что позволяет обработать все зависимые ячейки за один проход.

Эта диаграмма наглядно показывает, как происходит инвалидация кэша с оптимизацией, обеспечивающей проход по ячейкам только один раз.

Добавление новой ячейки, гарантирующее консистентное состояния таблицы?

graph TD

A[Начало: Sheet::SetCell(pos, text)] --> B[Проверка pos.IsValid()];

B -- Нет --> C[Выброс InvalidPositionException];

B -- Да --> D[Получение ячейки cell = GetCell(pos)];

D --> E[Вызов cell->Set(text)];

E --> F[Получение new\_dependencies из cell];

F --> G[Проверка HasCircularDependencies(pos, new\_dependencies)];

G -- Да --> H[Выброс CircularDependencyException];

G -- Нет --> I[Вызов UpdateDependencies(pos, new\_dependencies)];

I --> J[Вызов InvalidateCache(pos)];

J --> K[Конец];

C --> K;

H --> K;

**Описание шагов**

1. **Начало:** Вызывается метод Sheet::SetCell(pos, text), где pos — позиция новой ячейки, text — текст ячейки.
2. **Проверка позиции:** Проверяется, является ли позиция pos корректной с помощью pos.IsValid().
   * Если позиция некорректна, выбрасывается исключение InvalidPositionException.
   * Если позиция корректна, выполняется шаг 3.
3. **Получение ячейки:** Получается ячейка cell по позиции pos с помощью GetCell(pos).
4. **Установка текста ячейки:** Вызывается метод cell->Set(text) для установки текста ячейки.
5. **Получение зависимостей:** Получаются новые зависимости new\_dependencies из ячейки cell.
6. **Проверка циклических зависимостей:** Проверяется, не приведет ли добавление новой ячейки к циклическим зависимостям с помощью HasCircularDependencies(pos, new\_dependencies).
   * Если циклические зависимости обнаружены, выбрасывается исключение CircularDependencyException.
   * Если циклические зависимости не обнаружены, выполняется шаг 7.
7. **Обновление зависимостей:** Обновляются зависимости в графе с помощью UpdateDependencies(pos, new\_dependencies).
8. **Инвалидация кэша:** Инвалидируется кэш зависимых ячеек с помощью InvalidateCache(pos).
9. **Конец:** Выполнение метода завершается.

Эта диаграмма наглядно показывает, как происходит добавление новой ячейки с проверками на корректность позиции и циклические зависимости, а также с обновлением графа зависимостей и инвалидацией кэша для обеспечения консистентного состояния таблицы.