Учреждение Российской Академии наук
Санкт-Петербургский академический университет —
Научно-образовательный центр нанотехнологий РАН

На правах рукописи

Диссертация допущена к
защите
Зав. кафедрой

_____

« » _____ 2014 г.

Диссертация
на соискание ученой степени
магистра

Тема «Синтаксический анализ исходного кода, содержащего инструкции препроцессора»

Направление: 010600.68 — Прикладные математика и физика

Магистерская программа: «Математические и информационные технологии»

Выполнил студент С.А. Савенко

Руководитель: С.С. Игнатов

Рецензент:
к.ф.-м.н., доцент Д.Ю. Булычев

Санкт-Петербург
2014 г.

Содержание

1 Введение

Для создания большинства средств для работы с исходным кодом — таких, как компиляторы, среды разработки, инструменты автоматического поиска ошибок, браузеры исходного кода необходим синтаксический анализатор — инструмент, позволяющий сопоставить входной текст с грамматикой языка с тем, чтобы получить дерево синтаксического разбора[1].

Задача разработки синтаксических анализаторов (парсеров) на сегодняшний день решается одним из следующих способов:

- написание парсера вручную;
- использование библиотек парсер-комбинаторов (Parsec[1], jParsec[2] и др.);
- использование средств автоматической генерации синтаксическх анализаторов (ANTLR[3], Bison[4], JavaCC[5] и др.);

Использование парсер-генераторов позволяет существенно снизить трудозатраты при разработке инструментов обработки естественных и искуственных языков.

Часто при разработке программных продуктов в исходный код на языке программирования добавляются инструкции препроцессора, позволяющие повысить выразительность используемого языка программирования посредством текстовых подстановок из других файлов, осуществения макроподстановок, использования условной компиляции. Непрепроцессированный код также может зависеть от переменных окружения, переданных препроцессору, — тогда в одном файле с исходным кодом может быть описано сразу несколько различных программ. Это используется для создания целых семейств программных продуктов с общей кодовой базой[2], а также для создания высоко конфигурируемых программных систем. Использование текстовых препроцессоров при разра-

[1] `http://hackage.haskell.org/package/parsec`

[2] `https://github.com/abailly/jparsec`

[3] `http://www.antlr.org/`

[4] `http://www.gnu.org/software/bison/`

[5] `https://javacc.java.net/`

ботке программного обеспечения не является экзотикой — в некоторых языках, например C, C++, C#, препроцессирование исходного кода является одним из этапов компиляции.

Применение препроцессора приводит и к появлению ошибок, проявляющихся только при компиляции в некоторых конкретных конфигурациях — это могут быть синтаксические ошибки, ошибки типизации, ошибки при разрешении зависимостей и многие другие типы ошибок. Это приводит к необходимости обработки инструкций препроцессора в инструментах анализа исходного кода.

Для разработки таких инструментов требуются синтаксические анализаторы, способные интерпретировать инструкции препроцессора для того, чтобы успешно распознавать конструкции целевого языка программирования. Таким парсерам необходимо обрабатывать случаи «прерывания» грамматических конструкций языка инструкциями препроцессора, производить макроподстановки, обрабатывать варианты программы, которые могут быть получены при различных конфигурациях препроцессора.

Синтаксические анализаторы для языка C, обладающие описанной функциональностью, описаны в работах [3] и [4]. В данной работе представлен инструмент для упрощения разработки таких синтаксических анализаторов.

2 Постановка и анализ задачи

Проблема синтаксического разбора непрепроцессированного кода поднимается в различных публикациях уже достаточно продолжительное время. Это связано в первую очередь с большой популярностью языка C и, соответственно, его препроцессора — cpp. Далее будет описана основная функциональность препроцессора cpp и подходы к анализу непрепроцессированного кода.

2.1 Описание функциональности препроцессора

Препроцессор cpp предоставляет следующую функциональность:

- Включение файлов с исходным кодом посредством директив #include;
- Объявление и использование макроопределений (#define);
- Условную компиляцию основанную на статических условиях (директивы #if, #ifdef, #ifndef, и т. д.).

Как видно из примера, приведенного в листинге 1, результатом обработки входного файла препроцессором могут являться различные варианты исходного кода на языке C.

Набор предзаданных макроопределений позволяет выбрать единственный вариант программы, который впоследствии подаётся на вход транслятору. Далее такой набор определений и окружение, используемое для разрешения имён включаемых файлов, будем называть конфигурацией препроцессора.

```
1 #ifdef ENABLE_LOGGING
2 #include <stdio.h>
3 #define LOG(WHAT) printf("%d:%s\n", __LINE__, WHAT)
4 #else
5 #define LOG(WHAT)
6 #endif
7
8 int main(int argc, char ** argv) {
9   LOG("Terminating...");
10   return 0;
11 }
```

Листинг 1: Условная компиляция

Ясно, что программист, редактируя исходный код с инструкциями препроцессора, может производить изменения одновременно в огромном количестве программ, получаемых в различных конфигурациях, что зачастую приводит к ошибкам [5]. Обнаружение ошибок и другие задачи, связанные с обработкой непрепроцессированного исходного кода требуют рассмотрения сразу нескольких возможных конфигураций препроцессора.

2.2 Обзор способов разбора непрепроцессированного кода

Существует несколько подходов к анализу непрепроцессированного кода с учётом конфигураций препроцессора.

1. Обработка конфигураций «по одной». Идея подхода в том, чтобы производить поиск ошибок и решать другие задачи обработки исходного кода в каждой конфигурации препроцессора по отдельности. Подход очень прост в реализации, он применяется как неявно — при сборке проектов с разными конфигурациями, так и явно — например, в средах разработки Xcode[6] и KDevelop[7].

[6] https://developer.apple.com/xcode/

[7] http://kdevelop.org/

Однако, использование этого подхода не позволяет охватывать проекты целиком, так как большие проекты содержат огромное количество конфигураций. Например, сборка ядра Linux[8] требует установки значений более 10000 макроопределений [4], что исключает возможность использования такого подхода для исчерпывающего анализа кода.

2. Применение эвристик, основанных на типичных случаях использования препроцессора. Подход состоит в том, чтобы частично интерпретировать инструкции препроцессора, получая некоторое представление исходного кода с информацией о его вариантах в разных конфигурациях.

   Использование такого метода не позволяет обрабатывать код полностью, так как обработка всех возможных вариантов взаимодействия инструкций препроцессора с конструкциями языка программирования не представляется возможной.

   Тем не менее, этот подход позволяет достичь удовлетворительных результатов в некоторых задачах. Так, например, в Eclipse CDT[9] — интегрированной среде разработки для языков C и C++ — реализована подсветка синтаксиса и рефакторинги, учитывающие некоторые случаи использования препроцессора.

3. Построение графа, содержащего исходный код для всех возможных конфигураций препроцессора. Идея этого метода состоит в том, чтобы построить абстрактные синтаксические деревья для всех возможных конфигураций препроцессора.

   Количество деревьев разбора для всех возможных конфигураций препроцессора может экспоненциально зависеть от количества переменных в конфигурации. По этой причине построение деревьев в явном виде становится невозможным, и используются некоторые способы неявного представления леса разбора.

[8] `https://www.kernel.org/`

[9] `http://www.eclipse.org/cdt/`

Одним из возможных таких представлений является направленный ациклический граф, в котором каждой возможной конфигурации препроцессора соответствует некоторый путь из истока в сток. Подобные структуры данных применяются для хранения результатов синтаксического анализа текстов на языках, содержащих неоднозначности [6].

Несмотря на очевидные преимущества третьего подхода над первыми двумя, он очень редко применяется на практике. Отчасти это связано с высокой трудоёмкостью его реализации.

2.3 Цель работы

При рассмотрении существующих подходов к решению задачи синтаксического анализа исходного кода, содержащего инструкции препроцессора, было замечено, что и в случае построения графа, содержащего исходный код для всех возмождных конфигураций препроцессора, и в случае разбора конфигураций «по одной», используется только знание о синтаксисе языка препроцессора, его семантике, а также синтаксисе языка программирования.

Это наблюдение привело к предположению о возможности создания инструмента, позволяющего производить синтаксический анализ исходного кода, содержащего инструкции препроцессора, используя парсер, либо грамматику языка программирования, парсер для языка препроцессора препроцессора и интерпретатор его инструкций.

Целью настоящей работы является создание такого инструмента.

2.4 Задачи работы

Для достижения поставленной цели необходимо решить следующие задачи:

- изучить существующие решения задачи синтаксического разбора непрепроцессированного кода;

- разработать алгоритм, позволяющий использовать готовый синтаксический анализатор, либо грамматику языка программирования, для синтаксического анализа исходного кода, содержащего инструкции препроцессора;
- реализовать разработанный алгоритм;
- реализовать синтаксический анализатор для подмножества языка Erlang с инструкциями препроцессора, использующий разработанный алгоритм.

3 Существующие решения

Этот раздел содержит подробности реализаций подхода к решению проблемы синтаксического анализа непрепроцессированного исходного кода, состоящего в построении графа, содержащего деревья синтаксического разбора для всех конфигураций препроцессора (см. описание подхода в подразделе 2.2).

Наиболее известными реализациями этого подхода являются SuperC [3] и TypeChef [4].

3.1 Общие идеи

Построение графа, содержащего деревья синтаксического разбора для всех конфигураций препроцессора, так или иначе включает в себя два этапа:

1. Частичное препроцессирование или лексический анализ, сохраняющий конфигурацию.

   Выполняется интерпретация инструкций препроцессора, с сохранением информации об условных ветвлениях, и лексический анализ;

2. Синтаксический анализ, сохраняющий конфигурацию.

   Используется результат выполнения предыдущего этапа для построения графа разбора, содержащего синтаксические деревья для всех конфигураций препроцессора;

3.1.1 Лексический анализ, сохраняющий конфигурацию

Работа, выполняемая на этом этапе, условно состоит из двух частей — интерпретации инструкций препроцессора и проведения лексического анализа входного текста. Условность здесь состоит в том, что эти части могут выполняться одновременно, если языком препроцессора используются те же лексемы, что и языком программирования.

Инструкции препроцессора интерпретируются аналогично тому, как это делает сам препроцессор, за тем исключением, что строятся вари-

анты выходных последовательностей одновременно для всех конфигураций. Более подробно работа лексического анализатора, сохраняющего конфигурацию описана в подразделе 3.2.1.1 на примере его реализации в проекте SuperC.

Результатом выполнения этого этапа является последовательность лексем и узлов условного ветвления. Узлы ветвления представляют собой наборы из подпоследовательностей лексем и узлов условного ветвления, предваренных так называемыми условиями наличия (presence conditions). Каждый такой набор внутри узла ветвления соответствует некоторой альтернативной ветви условной компиляции в исходном непрепроцессированном коде.

3.1.2 Синтаксический анализ, сохраняющий конфигурацию

Этот этап состоит в преобразовании входной последовательности из лексем и узлов условного ветвления в дерево разбора, содержащее вершины, соответствующие конструкциям языка программирования, как в обыкновенном абстрактном синтаксическом дереве, а также вершины условного ветвления, каждая ветвь которых соответствует некоторой альтернативной ветви условной компиляции.

Появление узла ветвления во входной последовательности лексем может приводить к различной интерпретации лексем и уже разобранных конструкций, предшествующих текущей позиции. Это может привнести существенные сложности при разборе альтернативных ветвей условной компиляции. Так, в листинге 2 выражение справа от знака присваивания может быть либо числовой константой, либо аддитивным выражением. Это потребует переноса места ветвления в позицию следующую за символом равенства.

```
1  int main(int argc, char ** argv) {
2    int x = 0
3  #ifdef RET_1
4    + 1
5  #endif
6    ;
7    return x;
8  }
```

Листинг 2: Пример различной интерпретации лексем, предшестующих блоку условной компиляции

Помимо проблемы выбора места ветвления, имеет место проблема выбора места слияния. Она состоит в том, что лексемы, следующие за узлом условного ветвления могут быть проинтерпретированы по-разному для каждой из альтернативных ветвей.

```
1  #ifdef USE_LONG
2  #define RETURN_TYPE long
3  #else
4  #define RETURN_TYPE int
5  #endif
6
7  RETURN_TYPE function() {
8    ...
9    /* function body */
10   ...
11   return 0;
12 }
```

Листинг 3: Выбор точки слияния подпарсеров оказывает существенное влияние на скорость разбора

В листинге 3 приведен пример кода, неудачный выбор точки слияния в котором, может привести к существенному увеличению количества времени, необходимого для разбора. Как видно, тип возвращаемого значения функции зависит от макроопределения USE_LONG, что приведет к порождению двух ветвей разбора. Если точка слияния выбрана неудачно, то тело функции, следующей далее, может быть разобрано многократно.

Слишком раннее ветвление, равно как и слишком позднее слияние

могут привести к многократному разбору одних и тех же участков кода, что приводит к существенному снижению производительности и увеличению объёма используемой памяти.

3.2 Синтаксический анализ непрепроцессированного кода на языке C

На языке программирования C с момента его создания было написано огромное количество программного обеспечения различной сложности — в том числе и наиболее сложные программные системы: ядра операционных систем, системы управления базами данных, и т. д.. В связи с этим имеется большой спрос на средства работы с исходным кодом на языке C — среды разработки, инструменты для рефакторинга и автоматического поиска ошибок, браузеры кода, что, в свою очередь, пробудило интерес исследователей к проблеме синтаксического анализа непрепроцессированного кода на языке C.

3.2.1 SuperC

Проект SuperC является частью проекта xtc[10] Нью-Йоркского Университета. В этом проекте успешно решается проблема синтаксического анализа кода на языке C с инструкциями препроцессора, что подтверждается успешным разбором кода ядра Linux для архитектуры x86.

SuperC использует описанный в подразделе 3.1 двухэтапный подход. Рассмотрим более подробно каким образом реализованы эти этапы.

3.2.1.1 Лексирование и препроцессирование Для того, чтобы сформировать единицу компиляции, препроцессору, сохраняющему конфигурацию необходимо обработать следующие инструкции препроцессора: объявления и удаления макроопределений, директивы включения файлов, вызовы макроопределений.

Для корректной обработки этих инструкций, препроцессору, среди прочего, необходимо производить разрешение имён макроопредлений.

[10]eXTensible Compiler — `http://cs.nyu.edu/rgrimm/xtc/`

Это решается при помощи поддержания таблицы макроопределений на протяжениип обработки входных файлов. В эту таблицу вносятся записи, соответствующие директивам препроцессора #define и #undef. При этом объявления равно как и удаления макроопределений могут появляться в различных ветвях статических условных блоков, что приводит к тому, что, в зависимости от конфигурации, один и тот же вызов макроопределения может быть проинтерпретирован совершенно по-разному — возможны не только различные варианты макроподстановок, но и сами вызовы. Поэтому записи в таблице макроопределений ассоциируются с условиями их наличия, используемыми впоследствии для разрешения имён в различных ветвях условной компиляции.

Листинг 4 иллюстрирует некоторые случаи различных вызовов и подстановок макроопределений. Так, в случае наличия в конфигурации препроцессора объявления OP_FUN, в двенадцатой строке будет произведена подстановка макроопределения без аргументов. Если же OP_FUN окажется необъявленным, но будет объявлен OP_ID — будет произведена подстановка макроопределения с одним аргументом. В случае, если оба этих объявления не будут присутствовать, лексема, соответствующая имени макроопределения должна быть пропущена, а лексемы, соответствующие возможным аргументам вызова, проанализированы на предмет наличия других вызовов макроопределений.

```
1  #ifdef OP_FUN
2  int __op_fun(int x) {
3    return x + 2;
4  }
5  #define OP __op_fun
6  #elif OP_ID
7  #define OP(X) X
8  #endif
9
10 int main(int argc, char ** argv) {
11   int x = 0;
12   x = OP(x);
13   return x;
14 }
```

Листинг 4: Различные вызовы макроопределения при различных конфигурациях препроцессора

Таким образом, различные варианты подстановки для одного и того же вызова макроопределения, равно как и различные вызовы макроопределений, полученные из одних и тех же лексем, порождают неявные блоки условной компиляции, которые должны быть обработаны как препроцессирующим лексером, так и парсером на следующем этапе точно таким же образом, как и явные блоки условной компиляции, заданные при помощи директив препроцессора #if, #ifdef, #ifndef и др..

Вызовы макроопределений могут также пересекать границы блоков условной компиляции, что приводит к необходимости специальной обработки таких ситуаций, состоящей в дубликации лексем, составляющих вызов макроопределения. В листинге 5 приведен пример такой ситуации. Лексемы, следующие за блоком #ifdef ... #endif, а именно, «x, 10)» могут являться частью вызова макроопределения в случае, если объявлено макроопределение MACRO_MUL.

```
1  #define MUL(X, Y) ((X) * (Y))
2
3  int multiply(int x, int y) {
4    return x * y;
5  }
6
7  int main(int argc, char ** argv) {
8    int x = 1;
9    x =
10 #ifdef MACRO_MUL
11   MUL(
12 #else
13   multiply(
14 #endif
15     x, 10);
16   return x;
17 }
```

Листинг 5: Вызов макроопределения пересекает границу блока условной компиляции

Как уже было сказано ранее, для корректной обработки статических условий необходимо поддерживать условия наличия тех или иных участков кода. При этом трансформация статических условий в условия наличия зачастую требует частичного вычисления булевых выражений. Разработчики SuperC используют в своём решении двоичные диаграммы принятия решений (см. [7]), а именно — библиотеку JavaBDD[11]. Статические условия трансформируются в двоичные диаграммы принятия решений следующим образом:

1. Числовые константы заменяются на true и false, в зависимости от их значения;

2. Свободные макроопределения (переменные конфигурации) заменяются на переменные в BDD;

3. Арифмитические выражения заменяются на переменные в BDD;

[11] http://javabdd.sourceforge.net/

4. Выражения, проверяющие объявлено ли макроопределение, такие, как defined(MACRO), #ifdef, #ifndef и пр., заменяются на дизъюнкцию условий наличия, при которых это макроопределение объявлено.

Это позволяет эффективно сравнивать условия наличия, а также выявлять недостижимые ветви условной компиляции.

Результатом работы препроцессирующего лексера является направленный ациклический граф, состоящий из лексем с условиями их присутствия. В SuperC этот граф представлен последовательностью лексем языка с дополнительными лексемами, обозначающими точки ветвления и границы ветвей условной компиляции.

3.2.1.2 Синтаксический анализ Парсер, применяемый в проекте SuperC, основывается на LR-анализаторе (см. [1]), генерируемом парсер-генератором Bison[12].

Работа LR-анализаторов строится на основе стека, содеражащего терминальные и нетерминальные символы грамматики. Каждый шаг алгоритма LR-анализатора представляет собой одно из четырех возможных действий:

1. Сдвиг — помещение лексемы из входного буффера на стек;
2. Редукция — замена нетерминалом одного или нескольких верхних элементов стека;
3. Принятие — успешное завершение разбора;
4. Отвержение — завершение разбора с ошибкой.

Действие выбирается в зависимости от входной лексемы и текущего состояния стека.

Разработчики SuperC аргументируют выбор именно LR-анализатора в качестве основы для парсера сохраняющего конфигурацию следующими пунктами:

[12] `http://www.gnu.org/software/bison/`

- Возможность использования LR парсер-генератора для создания LR-таблиц, что значительно упрощает разработку;
- Удобство использования состояния парсера для обработки ветвей условной компиляции: достаточно использовать персистентный стек, реализованный на основе односвязного списка, в качестве стека LR-анализатора;
- Поддержка LR-парсерами широкого класса языков программирования.

Сам же FMLR-анализатор (Fork Merge LR parser) представляет собой алгоритм похожий по структуре на планировщик задач, где задачами выступают LR-подпарсеры. Алгоритм поддерживает очередь парсеров с соответствующими им условиями наличия и состояниями чтения входного буфера.

На каждом шаге алгоритма из очереди вынимается один парсер, для которого, в зависимости от следующего символа и состояния парсера выполняется одно из следующих действий:

- LR-действие, выбранное подпарсером и помещение подпарсера в конец очереди;
- Если следующим входным символом является точка ветвления, — создание подпарсеров для каждой из следующих возможных лексем, помещение этих подпарсеров в конец очереди.

После выполнения действия очередь парсеров исследуется на возможность слияния нескольких подпарсеров в один — это возможно лишь в том случае, если парсеры находятся на одном и том же месте во входном буфере, их LR-стеки совпадают, а также элементы на вершинах стеков соответствуют нетерминалам, для которых возможны ветвления и слияния. Каждое подмножество парсеров, для которого верны указанные условия, заменяется одним парсером, условие наличия которого выражено дизъюнкцией условий наличия парсеров из подмножества.

Стоит заметить, что выбор подмножества нетерминалов, для которых возможны ветвления и слияния, хотя и позволяет описать вид синтаксического дерева, явно указав синтаксические единицы, которые могут быть заменены на вершины-ветвления, привносит также некоторые сложности: отсутствие аннотаций для нетерминальных символов с продукциями некоторых видов может привести к порождению экспоненциального количества подпарсеров.

В качестве примера таких нетерминальных символов можно привести struct_declaration_list из грамматики языка C[13] — этот символ соответствует списку объявлений членов структуры. Часто в программах на языке C можно встретить условные объявления членов структур (см. листинг 6) — подобные объявления можно увидеть, например, в коде ядра Linux[14]. Если struct_declaration не будет проаннотирована, как синтаксическая единица, которая может быть заменена вершиной-ветвлением, то слияние подпарсеров порожденных на второй строке листинга 6 может быть осуществлено только после строки 11, где завершается struct_declaration_list, а это значит, что в каждом из двух подпарсеров будут порождены подпарсеры на строке 7.

```
1  struct cond_members {
2  #ifdef SM_A
3     int a;
4  #else
5     int b;
6  #endif
7  #ifdef SM_C
8     int c;
9  #else
10    int d;
11 #endif
12 };
```

Листинг 6: Условные объявления членов структуры

[13] http://www.lysator.liu.se/c/ANSI-C-grammar-y.html#struct-declaration-list

[14] В Linux 3.14 обявление структуры task_struct, описывающей поток исполнения, содержит условные объявления членов: http://lxr.free-electrons.com/source/include/linux/sched.h#L1164

3.2.2 TypeChef

TypeChef — исследовательский проект, объединивший усилия исследователей из многих университетов, имеющий своей целью поиск ошибок, вызванных интенсивным использованием препроцессора, в программах на языке C.

В основе алгоритма разбора непрепроцессированного исходного кода так же, как и в случае SuperC, лежит двухэтапный подход, поэтому далее будут описаны только особенности, отличающие TypeChef от SuperC.

3.2.2.1 Лексирование и препроцессирование Функциональность препроцессирующего лексера, сохраняющего конфигурацию, реализованного в проекте TypeChef аналогична функциональности, реализованной в аналоге — SuperC (см. подраздел 3.2.1.1).

Любопытной особенностью реализации препроцессирующего лексера в TypeChef является представление результата в виде последовательности лексем, где каждая лексема предварена условием её наличия. При этом отстутсвуют явные указания начал и концов ветвей условной компиляции — при синтаксическом анализе альтернативы и пропуски вычисляются при помощи сравнения условий наличия и решения задачи выполнимости булевых формул [8].

Задача выполнимости булевых формул в TypeChef решается с помощью библиотеки Sat4j[15].

3.2.2.2 Синтаксический анализ Авторы проекта TypeChef предлагают использовать библиотеку парсер-комбинаторов для реализации синтаксических анализаторов, поддерживающих условную компиляцию.

Основной идеей этой библиотеки является сокрытие логики обработки ветвей условной компиляции в нексольких парсер-комбинаторах, которые могут быть использованы для создания парсеров, позволяющих производить синтаксический анализ, учитывающий одновременно несколько ветвей условной компиляции.

[15] `http://www.sat4j.org/`

Более подробно о парсер-комбинаторах, предлагаемых в этом решении, можно прочесть в [9].

Такие библиотеки парсер-комбинаторов были реализованы на языках Haskell и Scala в рамках проекта TypeChef. Версия библиотеки, написанная на Scala, используется в TypeChef для реализации синтаксических анализаторов для языков C и Java, поддерживающих условную компиляцию.

3.3 Возможности адаптации решений к другим языкам

Несмотря на то, что изученные проекты TypeChef и SuperC были изначально нацелены на синтаксический анализ языка C с инструкциями препроцессора cpp, они предоставляют возможности для распространения используемых подходов на другие языки.

Однако, имеются препятствия к такому их использованию: может потребоваться реализация препроцессирующего лексера «с нуля», специальные обозначения для мест возможных условных ветвлений, модификации грамматики целевого языка, связанные с поддержкой библиотекой определенных классов языков, и другие.

3.3.1 TypeChef

Для того, чтобы использовать TypeChef для синтаксического анализа кода на языке отличном от C, требуется описать целевой язык, используя библиотеку парсер-комбинаторов на языке Scala.

Описание целевого языка при помощи парсер-комбинаторов вызвать некоторые проблемы, связанные, например, с использованием леворекурсивных правил в грамматике: правила вида $A \to Aa|a$ не могут быть переписаны явно — потребуется либо написание праворекурсивной версии этого правила с последующей трансформацией полученного поддерева, либо аналогичная конструкция — такая, как, например, в листинге 7.

```
1 class LeftRecursiveAParser extends MultiFeatureParser {
2   def A: MultiParser[A] = rep1(a) ^^ {
3     case first :: rest => rest.foldLeft(new A(first)) {
4       (l, r) => new A(l, r)
5     }
6   }
7 }
```

Листинг 7: Описание леворекурсивных правил с парсер-комбинаторами TypeChef

Для использования этих парсер-комбинаторов требуется хорошее представление об их работе и о структуре синтаксического дерева, которое разработчик хочет получать в качестве результата работы такого парсера. В случае неудачного выбора мест ветвления и слияния, производительность такого парсера может существенно деградировать.

3.3.2 SuperC

Серьёзных усилий требует и адаптация к другому языку программирования синтаксического анализатора из проекта SuperC. Для реализации синтаксического анализатора языка программирования с инструкциями препроцессора, SuperC требует от разработчика следующее:

- Описание лексера со специальными аннотациями, указывающими какие лексемы являются запятыми, скобками, решёткой, двойной решёткой (эти лексемы используются препроцессором языка C);
- Описание грамматики языка для парсер-генератора Bison;
- Аннотации к нетерминальным символам грамматики, указывающие, следует ли парсеру производить ветвления и слияния на узлах, соответствующих продукциям для этого нетерминального символа;
- Реализация семантических действий при разборе той или иной синтаксической конструкции — например, построение узлов AST.

Хочется заметить, что это достаточно большой объём работы, которая требует не только квалификации разработчика в области LR-анализаторов, но и хорошей степени знакомства с проектом SuperC.

4 Синтаксический анализ непрепроцессированного кода

В этом разделе описывается алгоритм, позволяющий производить синтаксический анализ исходного кода, представленного в виде графа лексем и точек ветвлений (см. подраздел 3.1.1).

4.1 Мотивация

Для построения графа синтаксического разбора, содержащего деревья разбора для всех возможных конфигураций препроцессора, необходимо знание о структуре языка, либо о синтаксическом анализаторе, используемом для разбора языка программирования.

В проектах SuperC и TypeChef использовалось знание о синтаксических анализаторах. А именно, SuperC использует LR-действия синтаксического анализатора языка программирования в явном виде, TypeChef же предлагает использовать свои инструменты для создания синтаксического анализатора языка программирования. Первое решение основывается на LR-анализаторах, второе — на LL, — и в первом и во втором случае может потребоваться модификация грамматики языка программирования, с тем, чтобы удовлетворить ограничения на обрабатываемые языки, накладываемые этими классами парсеров.

Результатом такого выбора стала высокая трудоёмкость адаптации этих решений для работы с другими языками программирования (см. подраздел 3.3).

Поэтому к алгоритму синтаксического анализа непрепроцессированного кода, разрабатываемому в настоящей работе, предъявляются следующие требования:

- Использование только грамматики языка программирования;
- Возможность синтаксического анализа исходного кода на любых языках, описываемых контекстно-свободными грамматиками [10].

Для удовлетворения данных требований был разработан универсальный алгоритм синтаксического анализа, основанный на алгоритме Earley [11].

4.2 Алгоритм Earley

Алгоритм синтаксического анализа Earley напрямую использует грамматику языка программирования и позволяет производить разбор текстов на языках, содержащих неоднозначности.

Результатом работы алгоритма является последовательность состояний, по которой можно определить принадлежность входного текста языку, а также восстановить все возможные деревья разбора. Далее будет описан сам алгоритм.

Как уже было замечено ранее, алгоритм строит последовательность некоторых состояний. Каждое состояние соответствует входной лексеме, для которой это состояние было получено и представляет собой упорядоченное множество элементов Earley (Earley item). Элемент Earley представляет собой продукцию грамматики с индексом в ней, обозначающимся точкой. Например, $A \to \cdot B$, $A \to B\cdot$, $A \to x \cdot B$. Часть продукции слева от точки — уже совпала со входом, а часть справа соответствует ожидаемому входу. Кроме этого, в элементе Earley сохраняется номер состояния первого появления этого элемента.

На каждом шаге алгоритм заполняет текущее состояние и инициирует следующее. Первое состояние алгоритма инициализируется элементами Earley, соответствующими начальному символу грамматики с точкой перед телом продукции. Рассмотрим теперь каким образом завершается состояние n и инициируется состояние $n+1$. Производятся три операции:

- Предсказание. Для каждого элемента Earley из состояния n, имеющего нетерминал после точки, в состояние n добавляются элементы Earley, соответствующие всем продукциям грамматики для этого нетерминала, с точками перед телами продукций.

- Сканирование. Для каждого элемента Earley из состояния n, имеющего терминал после точки, совпадающий с входной лексемой на

этом шаге, в состояние $n+1$ добавляется элемент Earley, с точкой сдвинутой за поглощенный терминал.

- Завершение. Эта операция производится для каждого элемента Earley из состояния $n+1$, в котором точка находится после тела продукции, — это соответствует окончанию сопоставления продукции тексту. Если элемент Earley соответствует продукции для некоторого нетерминала A, то производится поиск продукций, имеющих точку перед этим нетерминалом в состоянии, соответствующем первому появлению текущего элемента. В состояние $n+1$ добавляются элементы Earley, соответствующие поглощенному нетерминалу A.

В листинге 8 приведена функция doEarleyStep, в которой производятся описанные выше действия.

```
1  def doEarleyStep(lexeme, chart) = {
2    val currentCol = chart.lastColumn
3    val nextCol = chart.newColumnt
4    predict(currentCol)
5    scan(lexeme, currentCol, nextCol)
6    complete(nextCol)
7  }
8
9  def predict(currentCol) =
10   currentCol.items.forEach(predictForItem(_, currentCol))
11
12 def predictForItem(item, currentColumn) = {
13   if (item.isComplete || item.nextSymbol.isTerminal) return
14   val productions = Grammar .productionsFor(item.nextSymbol)
15   productions.forEach(production => {
16     val item = currentColumn.addItem(production)
17     if (item != null) predictForItem(item)
18   })
19 }
20
21 def scan(lexeme, currentCol, nextCol) =
22   currentCol.items.filter(_.expects(lexeme))
23     .forEach(nextCol.addItem(_.consume(lexeme)))
24
25 def complete(nextCol) =
26   nextCol.items.forEach(completeForItem(_, nextCol))
27
28 def completeForItem(item, nextCol) = {
29   if (!item.isComplete) return
30   item.startColumn.items.filter(_.expects(item))
31     .forEach(startColItem => {
32       val newItem = nextCol.addItem(startColItem.consume(item))
33       if (newItem != null)
34         completeForItem(newItem, nextCol)
35     })
36 }
```

Листинг 8: Псевдокод шага алгоритма Earley

Так, результатом работы алгоритма станет последовательность из $N+1$ состояния для входного текста длины N лексем. Если входной текст

принадлежит языку, грамматика которого использовалась для разбора, то в последнем состоянии будет находиться хотя бы один завершенный элемент Earley, впервые добавленный в первом состоянии, соответствующий начальному символу грамматики.

Описанный выше алгоритм производит только распознавание входа, т. е. позволяет определить принадлежит ли входной текст языку, описываемому используемой грамматикой. Несколько модифицировав алгоритм распознавания, можно получить средство для восстановления деревьев разбора.

Для этого необходимо при выполнении сканирования и завершения сохранять указатели/ссылки на элементы Earley, из которых мог быть получен новый элемент Earley (это может быть реализовано в функции consume, используемой в листинге 8).

Обратный проход по этим указателям, начиная с состояния, соответствующего завершенному разбору для начального символа, будет соответствовать одному из возможных деревьев разбора. Более подробно о преобразовании алгоритма распознавания в парсер можно прочесть в оригинальной работе [11], а также в [12].

Замечательная особенность алгоритма состоит в том, что он работает с любой контекстно-свободной грамматикой и не требует её приведения к какому-либо специальному виду.

4.3 Модификация алгоритма Earley

Основная идея алгоритма состоит в том, чтобы для различных ветвей условной компиляции строить различные последовательности состояний, сохраняя условия наличия, после чего объединять полученные наборы состояний.

Далее описывается модификация алгоритма Earley, позволяющая работать с графом, полученным на выходе препроцессирующего лексера, сохраняющего конфигурацию, а также необходимые модификации алгоритма восстановления графа разбора..

4.3.1 Распознавание

Алгоритм начинает свою работу точно как оригинальный алгоритм распознавания Earley.

Если в процессе работы встречается точка ветвления в графе лексем, то создаются новые распознаватели Earley для каждой из ветвей условной компиляции. В каждой ветви используется состояние, соответствующее состоянию перед точкой ветвления, в качестве начального. Далее этот же алгоритм рекурсивно обрабатывает каждую ветвь независимо (это предоставляет возможность для распараллеливания алгоритма). По завершении обработки ветвей, производится слияние последнего состояния каждой ветви в новое состояние и работа алгоритма продолжается. (см. Листинг 9)

```
1  def recognize(stream) : Chart = {
2    val chart = new Chart
3    processStream(stream, chart)
4    predict(chart.lastColumn)
5    complete(chart.lastColumn)
6    chart
7  }
8
9  def processStream(stream, chart) = {
10   var lexeme = stream.eof
11   while ((lexeme = stream.next) != stream.eof) {
12     lexeme match {
13       case Fork(streams) => {
14         val colBeforeFork = chart.lastColumn
15         predict(colBeforeFork)
16         val forkLastColumns = streams.map(
17           forkStream => {
18             val subChart = chart
19               .subChart(forkStream.presenceCondition)
20             processStream(forkStream, subChart)
21             subChart.lastColumn
22           }
23         )
24         val colAfterFork = chart.newColumn()
25         forkLastColumns.forEach(colAfterFork.addItemsFrom(_))
26         val orOfForkPCs = streams
27           .foldLeft(PresenceCondition.False)
28             (_ or _.presenceCondition)
29         if (!orOfForkPCs.isTrue)
30           colAfterFork
31             .addItemsFrom(colBeforeFork, orOfForkPCs.not)
32       }
33       case lexeme => doEarleyStep(lexeme, chart)
34     }
35   }
36 }
```

Листинг 9: Псевдокод алгоритма распознавания

Стоит заметить, что теперь элемент Earley может быть получен не только из разных предшествующих элементов Earley, но и из одних и

тех же предшественников путями, отличающимися условиями наличия элементов в них.

Для корректной обработки таких случаев в элементах Earley дополнительно для каждого предшественника элемента хранится условие наличия и множество редукций, по которым этот элемент мог быть получен из предыдущего.

Под редукцией здесь понимается завершенный элемент Earley, либо терминал, находящийся слева от точки в текущем элементе Earley. В редукции также сохраняется указатель на состояние Earley, в котором был начат разбор этой редукции. Это позволяет восстанавливать все возможные способы получения данного элемента Earley, что используется при восстановлении графа разбора.

Условия наличия для элемента Earley вычисляется по-разному в зависимости от способа получения элемента. Для элементов, полученных предсказанием это условие наличия предшественника. Для элементов, полученных сканированием это логическое «И» условия наличия терминала и условия наличия предшественника. Для элементов, полученных завершением это логическое «И» условия наличия предшественника и условия наличия элемента Earley, использованного для завершения.

Соответствующие изменения для процедур алгоритма Earley приведены в листинге 10.

```
1  def predictForItem(item, currentColumn) = {
2    if (item.predictionIsDone || item.isComplete ||
3      item.nextSymbol.isTerminal) return
4    item.predictionIsDone = true
5    val pc = getOrOfPrsenceConditions(item, currentColumn)
6    val productions = Grammar.productionsFor(item.nextSymbol)
7    productions.forEach(production => {
8      val item = currentColumn.addItem(production, pc)
9      if (item != null)
10       predictForItem(item)
11   })
12 }
13
14 def scan(lexeme, currentCol, nextCol) =
15   currentCol.items.filter(_.expects(lexeme))
16     .forEach(item => {
17       val pc = item.presenceCondition and
18         lexeme.presenceCondtion
19       nextCol.addItem(item.consume(lexeme), pc)
20     })
21
22 def completeForItem(item, nextCol) = {
23   if (!item.isComplete) return
24   item.startColumn.items filter(_.expects(item))
25     .forEach(startColItem => {
26       val pc = item.presenceCondition and
27         startColItem.presenceCondition
28       val newItem =
29         nextCol.addItem(startColItem.consume(item), pc)
30       if (newItem != null)
31         completeForItem(newItem, nextCol)
32     })
33 }
```

Листинг 10: Псевдокод модифицированных процедур алгоритма Earley

Как и в случае оригинального алгоритма, наличие в последнем состоянии завершенного элемента Earley, впервые появившегося в первом состоянии, будет соответствовать принадлежности исходного текста языку. Однако, следует заметить, что наличие таких элементов в последнем

состоянии не означает корректности кода во всех возможных конфигурациях — ошибки разбора, связанные с ветвлениями следует отслеживать отдельно.

4.3.2 Построение графа синтаксического разбора

Алгоритм восстановления деревьев разбора, описанный в [12], позволяет восстанавливать все возможные деревья разбора для текстов на языках, содержащих неоднозначности. Алгоритм восстановления графа синтаксическго разбора, описанный далее, предлагает аналогичный способ разбора, позволяя отслеживать условия наличия тех или иных вариантов разбора.

Суть алгоритма восстановления графа синтаксического разбора состоит в преобразовании завершенных элементов Earley в узлы графа-результата. Такое преобразование осуществляется следующим образом:

- Если у элемента единственный предок, то создаётся вершина графа, соответствующая этому элементу Earley, дети этой вершины строятся для этого единственного предка;
- Если у элемента несколько предков, то производится разрешение неопределенностей (опционально). После этого шага создаётся узел ветвления, каждая ветвь которого — подграф, построенный для одного из предков с его условием наличия, условие наличия узла ветвления — логическое «ИЛИ» условий наличия его ветвей.

Подграф для предка элемента Earley строится так:

1. Создаётся вершина, помеченная нетерминальным символом элемента Earley;
2. Строятся подграфы для детей этой вершины;
3. Условие наличия этой вершины есть конъюнкция условия наличия элемента Earley и логического «И» всех детей этой вершины.

Построение подграфов для детей вершины производится рекурсивно следующим образом:

- Если этот элемент мог быть получен из предшественника единственной редукцией, то строится узел для этой редукции, последующие узлы строятся рекурсивно;
- Если этот элемент мог быть получен из предшественника несколькими редукциями, то создаётся узел ветвления, каждая ветвь которого соответствует различным вариантам разбора начиная с этого потомка. Каждая ветвь строится рекурсивно.

Для функции, преобразующей завершенные элементы в узлы графа-результата разумно применять мемоизацию, так как в некоторых случаях они могут быть вызваны многократно для одних и тех же узлов.

Таким образом, модифицированный алгоритм Earley позволяет строить граф, содержащий синтаксические деревья для всех возможных конфигураций препроцессора без какой-либо дополнительной информации о возможных точках ветвления и слияния ветвей условной компиляции.

5 Реализация библиотеки для создания синтаксических анализаторов

Реализация библиотеки для создания синтаксических анализаторов исходного кода, содержащего инструкции препроцессора, имеет следующие цели:

- Проверить возможность применения алгоритма, описанного в подразделе 4.3;
- Предоставить удобный способ реализации синтаксических анализаторов исходного кода, содержащего инструкции препроцессора, лишенный недостатков аналогов, описанных в 3.3.

Библиотека была реализована на языке Java и имеет в своей основе двухэтапный подход, описанный в подразделе 3.1.

5.1 Препроцессирующий лексер

Первым шагом, необходимым для анализа исходного кода, содержащего инструкции препроцессора, является частичное препроцессирование и выделение лексем. Алгоритм работы препроцессирующего лексера, реализованного в этой библиотеке описан в подразделе 3.1.1.

Препроцессирующий лексер абстрагируется от конкретного используемого препроцессора при помощи интерфейсов для парсера языка препроцессора и узлов AST языка препроцессора, включающего в себя узлы, соответствующие:

- Условной компиляции;
- Объявлению макроопределения;
- Удалению объявления макроопределения (#undef);
- Включению файла;
- Любым другим типам узлов.

Помимо этого, выделены интерфейсы для лексера языка программирования, парсеров вызовов макроопределений, файлов с исходным кодом (они используются для разрешения имён включеных файлов), условий наличия.

Такой набор абстракций позволяет избавить пользователя библиотеки от необходимости самостоятельно реализовывать препроцессирующий лексер, сохраняющий конфигурацию.

Результатом работы препроцессирующего лексера является граф лексем и ветвлений, соответствующих явным ветвям условной компиляции.

5.2 Парсер, сохраняющий конфигурацию

Как было замечено ранее, алгоритм, описанный в подразделе 4.3, использует только грамматику языка для синтаксического анализа графа лексем, полученного на предыдущем этапе разбора. Разработанная библиотека позволяет описывать грамматику языка, используя нотацию BNF, описанную в [10], что приводит к возможности простого и интуитивно понятного описания грамматики языка (см. Листинг 11).

```
1  public final class PlusLanguage {
2    //terminals
3    public static final Integer INT = 0;
4    public static final Integer PLUS = 1;
5    //nonterminals
6    public static final String EXPR = "expression";
7
8    private PlusLanguage() {
9    }
10
11   public static EarleyGrammar createGrammar() {
12     return EarleyGrammarBuilder.grammar(EXPR)
13
14         .rule(EXPR).terminal(INT)
15               .completeRule()
16
17         .rule(EXPR).nonTerminal(EXPR)
18               .terminal(PLUS)
19               .terminal(INT)
20               .completeRule()
21
22         .build();
23   }
24 }
```

Листинг 11: Пример описания грамматики простого языка.

Используя такое описание грамматики входного языка и входной граф лексем, алгоритм строит абстрактное синтаксическое дерево, содержащее узлы нескольких типов:

- Альтернативы — соответствует точке ветвления для различных вариантов разбора;
- Условная ветвь — соответствует варианту синтаксического разбора для некоторой ветви условной компиляции;
- Лист — соответствует терминальным символам;
- Композит — соответствует нетерминальным символам.

Основное отличие от синтаксических деревьев, применяемых при синтаксическом анализе исходного кода, не содержащего инструкции препроцессора, состоит в том, что вместо любой последовательности дочерних вершин вершины композитного типа может присутствовать вершина типа альтернативы. Таким образом, выбирая обход для нужных условий наличия, можно получить дерево разбора, которое было бы получено для некоторой конкретной конфигурации препроцессора.

6 Синтаксический анализ кода на языке Erlang

Для проверки работоспособности разработанной библиотеки для создания синтаксических анализаторов исходного кода, содержащего инструкции препроцессора, было предложено реализовать синтаксический анализатор для подмножества языка Erlang.

Синтаксический анализатор для подмножества языка Erlang, созданный с помощью разработанной библиотеки, является прототипом синтаксического анализатора, который будет использован в проекте intellij-erlang[16] — интегрированной среде разработки для языка Erlang на основе IntelliJ IDEA[17].

На данный момент в intellij-erlang используется синтаксический анализатор, сгенерированный при помощи Grammar-Kit[18]. Инструкции препроцессора в этом анализаторе включены в грамматику языка с тем, чтобы обеспечить возможность разбора наиболее часто встречающихся паттернов использования препроцессора. Однако, такой подход не позволяет производить анализ кода, содержащего некоторые варианты использования препроцессора, а также производить поиск ошибок, связанных с подстановкой макроопределений и условной компиляцией.

В этом разделе описываются особенности реализации синтаксического анализатора исходного кода на подмножестве языка Erlang, содержащем инструкции препроцессора, с использованием библиотеки, описанной в разделе 5.

6.1 Препроцессор

Здесь будет приведено краткое описание функциональности препроцессора языка Erlang (за полным описанием следует обратиться к [13]) а также особенностей её реализации с использованием разработанной библиотеки.

[16] `https://github.com/ignatov/intellij-erlang`

[17] `http://www.jetbrains.com/idea/`

[18] `https://github.com/JetBrains/Grammar-Kit`

6.1.1 Включение файлов

Препроцессор языка Erlang позволяет производить подстановку текстовых файлов, имена которых разрешаются при помощи двух разных способов:

- Имя файла, включенного при помощи аттрибута include, разрешается относительно каталога, из которого этот файл был подключен;
- Имя файла, включенного при помощи аттрибута include_lib, разрешается относительно набора каталогов, содержащих зависимости данного проекта.

Оба способа легко реализуются в рамках интерфейсов, предоставляемых библиотекой. Непосредственно подстановка текста реализована в самой библиотеке и не потребовала каких-либо усилий.

6.1.2 Условная компиляция

Для реализации функциональности, связанной с условной компиляцией, такой как условное ветвление и подстановки макроопределений, библиотека требует реализации условий наличия.

Условия наличия в случае Erlang представляют собой формулы логики высказываний, где переменным соответствуют утверждения об объявленности макроопределений. Формулы в реализации препроцессора для Erlang хранятся в дизъюнктивной нормальной форме.

6.1.3 Макроопределения

Абстракции, предоставляемые библиотекой, позволили без труда реализовать объявления и разобъявления макроопределений, а также парсеры их вызовов, что позволило получить работающую подстановку макроопределений.

6.2 Лексер

В качестве лексера языка Erlang был использован несколько модифицированный лексер из проекта intellij-erlang, генерируемый при помощи сканнер-генератора JFlex[19].

Для использования сгенерированного лексера понадобилось только реализовать адаптер (см. [14]) для того, чтобы сгенерированный лексер реализовывал интерфейс, предоставляемый библиотекой.

6.3 Парсер

Реализация парсера средствами разработанной библиотеки состояла в переписывании правил грамматики языка Erlang[20], описанной в BNF, на предметно-ориентированном языке, пример использования которого приведен в листинге 11.

Никаких действий по адаптации грамматики для использования с разработанной библиотекой не потребовалось, — переписанные правила имеют тот же вид, что и правила исходной грамматики.

[19] `http://jflex.de/`

[20] `https://github.com/erlang/otp/blob/master/lib/stdlib/src/erl_parse.yrl`

7 Анализ результатов

В этом разделе описываются результаты, полученные в настоящей работе, описываются преимущества и недостатки разработанного решения, проводится сравнение с аналогичным решением из проекта TypeChef, описываются возможные улучшения предложенного решения.

7.1 Особенности разработанного решения

Одним из результатов настоящей работы является модификация алгоритма Earley (см. подраздел 4.3), позволяющая производить синтаксический анализ для контекстно-свободных языков, используя последовательность лексем, содержащую условные ветвления, в качестве входных данных. Этот алгоритм был реализован в библиотеке для создания синтаксических анализаторов, описанной в разделе 5.

Для создания синтаксического анализатора, поддерживающего обработку ветвлений условной компиляции, разработчику необходимо лишь описать на предметно-ориентированном языке, предоставляемом библиотекой, грамматику используемого языка программирования. В силу поддержки библиотекой всех контекстно-свободных языков, разрабочику не придётся модифицировать грамматику языка программирования. Это выгодно отличает разработанное решение от аналогов, генерирующих LL и LR анализаторы.

Следует заметить, что предлагаемое решение не требует от разработчиков указания синтаксических конструкций, в которых возможно наличие вершин, соответствующих точкам условного ветвления, а подразумевает возможность ветвления во всех продукциях входной грамматики. Таким образом, язык программирования описывается совершенно так же, как если бы он описывался для парсер-генератора без поддержки точек условного ветвления.

В проекте TypeChef, например, где выбран другой подход, неудачное выделение мест ветвления может привести к сокращению количества способов использования препроцессора, а также к сложности алгоритма разбора, экспоненциально зависящей от количества точек условного

ветвления.

7.2 Сравнение с TypeChef

В разделе 6 описывается применение разработанного решения для синтаксического анализа подмножества языка Erlang, содержащего инструкции препроцессора. В этом подразделе описывается опыт применения проекта TypeChef для создания синтаксического анализатора для того же подмножества языка Erlang, проводится эмпирическое сравнение объёма трудозатрат на реализацию решений, производительности полученных решений.

7.2.1 Лексический анализ

Проект TypeChef содержит в себе реализацию препроцессирующего лексера, сохраняющего информацию о ветвях условной компиляции, для языка C, а также предоставляет интерфейсы для альтернативных его реализаций. Эти интерфейсы содержат описание функциональности специфичной для препроцессора языка C, что препятствует их использованию для других языков.

Однако, часть функциональности препроцессора языка C, реализованная в TypeChef может быть переиспользована — например, набор средств для работы с булевыми формулами, описанный в подпроекте FeatureExprLib[21].

Таким образом, для реализации лексического анализаторв в TypeChef потребуется практически полностью реализовывать препроцессирующий лексер.

Благодаря тому, что парсеры в TypeChef не имеют явной зависимости от лексеров, а используют последовательности лексем, где каждой лексеме соответствует условие её наличия, реализовывать препроцессирующий лексер «с нуля» не потребовалось: был реализован адаптер, позволяющий переиспользовать лексер, описанный в подразделе 5.1

[21] `https://github.com/ckaestne/TypeChef/tree/master/FeatureExprLib`

Реализация лексера, осуществляющего препроцессирование, сохраняющее ветвления условной компиляции, с помощью решения, предлагаемого в настоящей работе, также требует существенных трудозатрат, однако, выгодно отличается наличием реализации таблицы макроопределений, разрешением имён макроопределений и другой функциональности, не имеющей зависимостей от конкретного языка программирования.

7.2.2 Синтаксический анализ

Для реализации синтаксического анализатора разработчики TypeChef предлагают применять реализованные ими парсер-комбинаторы, описывающие возможные места появления ветвлений условной компиляции.

При использовании этой библиотеки пришлось несколько изменить грамматику языка Erlang с тем, чтобы избавиться от леворекурсивных правил в грамматике. Леворекурсивные конструкции пришлось реализоваывать вручную — в листинге 12 приведен код, использованный для разбора леворекурсивных бинарных операций.

```
1 def leftAssocBop(argParser: MultiParser[Expr],
2                   opParser: MultiParser[Elem]) =
3   argParser ~ repPlain(opParser ~ argParser) ^^ {
4     case left ~ tail => tail.foldLeft(left) {
5       case (l, op ~ r) =>
6         new BinaryExpr(l, r, op.getType)
7     }
8   }
```

Листинг 12: Реализация леворекурсивных конструкций в TypeChef

Подобных модификаций при использовании решения, предлагаемого в данной работе не потребовалось.

7.2.3 Объём исходного кода

Напомним, что TypeChef использует язык Scala для описания синтаксических анализаторов, в то время, как решение, предлагаемое в данной работе использует Java.

Для описания грамматики используемого подмножества языка Erlang

для TypeChef понадобилось более 200 строк исходного кода, в то время, как для библиотеки, разработанной в рамках данной работы, ∼150 строк исходного кода.

Из этого можно заключить, что разработанное решение обладает более лаконичным синтаксисом для описания грамматики языка программирования. Любопытно также отметить, что Scala — более выразительный язык, чем Java, и позволяет писать до двух раз более короткие программы, обладающие схожей функциональностью [15].

7.2.4 Производительность

Для эмпирического сравнения производительности применялся набор входных данных[22], используемый для тестирования работы синтаксического анализатора, описанного в разделе 6.

Так как для обоих решений использовался один и тот же лексический анализатор, время, затраченное на его работу, а также адаптацию результатов его работы к интерфейсам TypeChef, не учитывалось при сравнении. После запуска анализатора для каждого входного файла выполнялась сборка мусора.

Из графика 1 видно, что решения показывают сравнимую скорость работы на большинстве тестов. Медианные значения составляют 25 и 45 миллисекунд для Earley и TypeChef, соответственно.

Тесты, в которых синтаксический анализатор, реализованный с помощью TypeChef, показывает значительно худшие результаты имеют следующее отличие — в них присутствуют арифметические выражения с альтернативными подвыражениями — один из таких тестов приведен в листинге 13.

[22]Файлы, использованные в качестве тестовых данных доступны по адресу: `https://github.com/deadok22/preprocessing-parser/tree/master/erlang/testData/earleyParser`

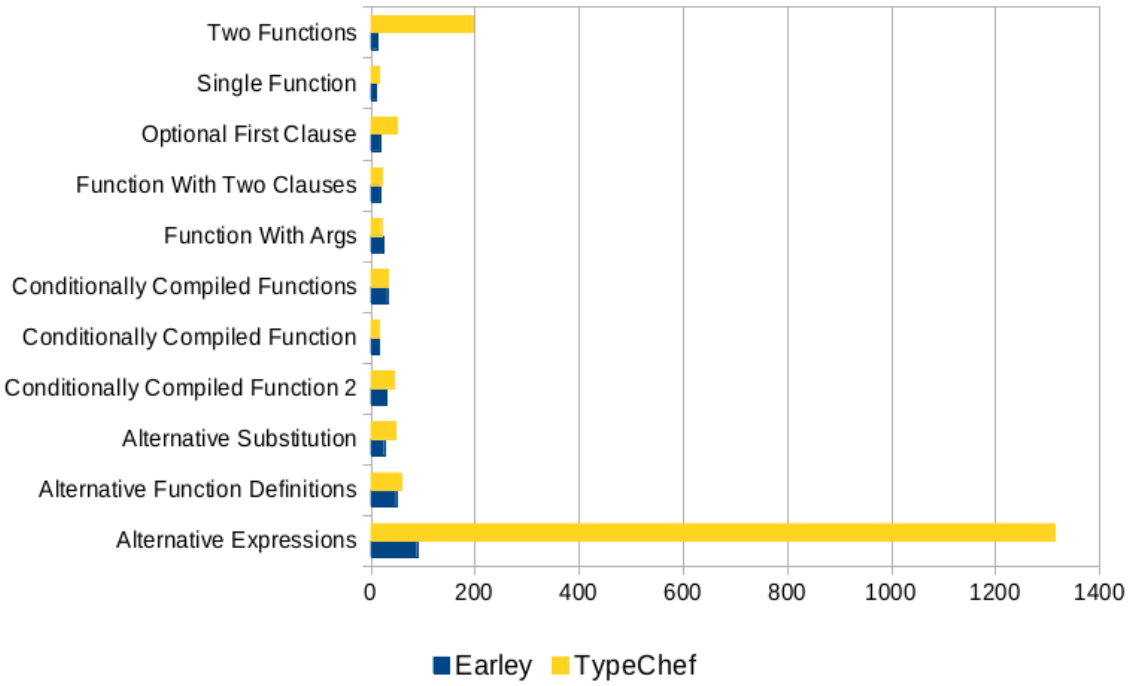

График 1: Время работы на тестовых данных (мс.)

```
1 -ifdef(X).
2 -define(EXPR, * 19).
3 -else.
4 -define(EXPR, + 18).
5 -endif.
6
7 foo() ->
8     (11 ?EXPR) - 94.
```

Листинг 13: Один из тестов с альтернативными подвыражениями

Низкая скорость разбора может быть связана с неудачным описанием списочных конструкций, что приводит к экспоненциальной сложности разбора таких выражений в TypeChef.

Из вышеописанного можно заключить, что разработанное решение не уступает TypeChef в производительности, требуя при этом значительно меньших усилий при описании входного языка.

7.3 Направления развития решения

В этом подразделе описываются возможные направления работы связанные с улучшением решения, предлагаемого в настоящей работе, и устранением имеющихся в нем недостатков.

На данный момент библиотека не позволяет использование в грамматике пустых правил. Известны модификации алгоритма Earley, позволяющие обрабатывать пустые правила [16] — они могут быть использованы и в модифицированном алгоритме, описанном в подразделе 4.3.

Выразительность предметно-ориентированного языка для описания грамматик входных языков также может быть увеличена при помощи поддержки EBNF синтаксиса.

Расширение функциональности предметно-ориентированного языка также возможно за счет применения аннотаций к правилам и за счет учета порядка их следования для разрешения неопределенностей при разборе. Это позволило бы описывать, например, грамматику выражений в интуитивно понятной форме, например, так:

$$Expr \ ::= \ Expr \ + \ Expr \ //left - associative$$

.

Модифицированный алгоритм Earley может быть улучшен при помощи обработки некоторых специальных случаев — таких, например, как списочные структуры в грамматиках языка. На данный момент, наличие во входном тексте списка элементов, элементы которого находятся в различных ветвях условной компиляции, приводит к экспоненциальной от количества альтернативных элементов списка зависимости количества требуемой оперативной памяти и времени восстановления графа синтаксического разбора. Этого можно избежать, допуская не только альтернативное, но и условное вхождение элементов в списочные структуры.

Разработанный алгоритм может быть исследован на наличие возможности поддержания контекстов при разборе ветвей условной компиляции, что позволило бы проводить синтаксический анализ текстов на контекстно-зависимых языках программирования.

Другим возможным направлением работы является разработка предметно-ориентированного языка для описания парсера и интерпретатора инструкций препроцессора — это позволило бы существенно сократить трудозатраты на разработку синтаксического анализатора исходного кода, содержащего инструкции препроцессора.

8 Заключение

Настоящая работа продемонстрировала возможность создания инструмента, позволяющего производить синтаксический анализ исходного кода, содержащего инструкции препроцессора, используя синтаксический анализатор исходного кода языка препроцессора, средства интерпретации его инструкций и грамматику языка программирования.

В работе предложена модификация алгоритма синтаксического анализа Earley, использование которой позволяет восстанавливать синтаксические деревья, соответствующие всем возможным конфигурациям препроцессора.

Предложенный алгоритм был реализован в библиотеке позволяющей создавать синтаксические анализаторы исходного кода, содержащего инструкции препроцессора. Библиотека была применена для синтаксического анализа исходного кода на подмножестве языка Erlang.

Было произведено сравнение разработанной библиотеки с аналогичным решением TypeChef — разработанное решение позволяет с меньшими трудозатратами реализовать синтаксический анализатор исходного кода, содержащего инструкции препроцессора, обладающий сравнимыми характеристиками производительности.

Использование подхода предложенного в этой работе позволяет существенно сократить время на разработку синтаксического анализатора исходного кода, содержащего инструкции препроцессора.

Полученные результаты могут быть использованы для разработки средств автоматической генерации синтаксических анализаторов для языков программирования, используемых с препроцессором.

Библиография

[1] Alfred V. Aho и др. *Compilers Principles Techniques, & tools*. second edition. `http://dragonbook.stanford.edu/`. 2006.

[2] D. Ganesan и др. "Verifying Architectural Design Rules of the Flight Software Product Line". в: *SPLC '09 Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University. 2009, с. 161—170.

[3] P. Gazzillo и R. Grimm. "SuperC: Parsing All of C by Taming the Preprocessor". в: *PLDI'12 Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. `http://cs.nyu.edu/~rgrimm/papers/pldi12.pdf`. ACM. 2012, с. 323—334.

[4] A.Kenner и др. "TypeChef: Toward Type Checking #ifdef Variability in C". в: *FOSD'10 Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*. `http://www.cs.cmu.edu/~ckaestne/pdf/FOSD10-typechef.pdf`. ACM. 2010, с. 25—32.

[5] M. Ribeiro и др. "On the Impact of Feature Dependencies When Maintaining Preprocessor-based Software Product Lines". в: *GPCE'11 Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*. `http://itu.dk/people/brabrand/gpce-2011.pdf`. ACM. 2011, с. 23—32.

[6] D. Grune и C. Jacobs. "Parsing Techniques: A Practical Guide". в: 2nd edition. Springer, 2007. гл. 3.7.3 Parse Forests.

[7] S. B. Akers. "Binary Decision Diagrams". в: *IEEE Transactions on Computers* 27 (июнь 1978), с. 509—516.

[8] M. Garey и D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[9] C. Kastner и др. "Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation". в: *OOPSLA'11 Proceedings of the 2011 ACM International Conference on Object Oriented Programming*

Systems Languages and Applications. http://www.cs.cmu.edu/~ckaestne/pdf/oopsla11_typechef.pdf. ACM, с. 805—824.

[10] K. Cooper и L. Torczon. "Engineering a Compiler". в: 2nd edition. 2011. гл. 3.2.2 Context-Free Grammars.

[11] Jay Earley. "An Efficient Context-Free Parsing Algorithm". http://reports-archive.adm.cs.cmu.edu/anon/anon/usr/ftp/scan/CMU-CS-68-earley.pdf. дис. ... док. Carnegie Mellon University, авг. 1968.

[12] D. Grune и C. Jacobs. "Parsing Techniques: A Practical Guide". в: 2nd edition. Springer, 2007. гл. 7.2.1.2 Constructing a Parse Tree.

[13] Ericsson AB. "Erlang/OTP System Documentation". в: Ericsson AB, 2014. гл. 5.9 The Preprocessor.

[14] E. Gamma и др. "Design Patterns: Elements of Reusable Object Oriented Software". в: Addison-Wesley, 2000, с. 139—150.

[15] V. Pankratius, F. Schmidt и G. Garreton. "Combining Functional and Imperative Programming for Multicore Software: an Empirical Study Evaluating Scala and Java". в: *ICSE'12 Proceedings of the 34th International Conference on Software Engineering*. IEEE. 2012, с. 123—133.

[16] D. Grune и C. Jacobs. "Parsing Techniques: A Practical Guide". в: 2nd edition. Springer, 2007. гл. 7.2.3 Handling ϵ-Rules.