Титульный лист материалов по дисциплине

ДИСЦИПЛИНА *Теория формальных языков*

ИНСТИТУТ *Информационных технологий*

КАФЕДРА *Вычислительной техники*

ВИД УЧЕБНОГО МАТЕРИАЛА *Лекция*

ПРЕПОДАВАТЕЛЬ *Унгер Антон Юрьевич*

СЕМЕСТР *3 семестр*

1. ЯЗЫКИ ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ

В области разработки программного обеспечения основным инструментом являются языки программирования высокого уровня. Разбором исходного текста занимаются трансляторы, или компиляторы – специальные программы, осуществляющие разбор структурированного текста и перевод его в некую высокоуровневую структуру.

Данная фаза – перевод – является первым этапом обработки. Он называется *анализом*. Данный этап включает в себя множество проходом, в результате которых формируется машинно-независимое представление программы в виде некоторой структуры данных.

Второй этап, называемый *синтезом*, занимается переводом программы из машинно-независимого представления в программу, понятную машине [1]. Данный этап сильно привязан к целевой машине, точнее архитектуре процессора, для которого генерируется целевой код.

Для программиста, пишущего программы на языке высокого уровня, компилятор представляется черным ящиком, инструментом, единственное что необходимо знать о котором – это синтаксис исходного языка. Более продвинутые разработчики пользуются дополнительными опциями компилятора, такими как уровни оптимизации, однако мало кто из практикующих программистов задумывается над тем, что из себя представляет программа компилятора внутри.

1.1. Появление высокоуровневых языков

С самого зарождения программирования, как новой области компьютерных технологий, стало понятно, что писать сложные программы на языке машинных кодов – сложный и чрезвычайно трудоемкий процесс, чреватый ошибками. Первым шагом к тому, чтобы упростить процесс разработки стало изобретение *ассемблера* – языка, который являлся промежуточным между человеком и машиной. Ассемблер давал возможность называть инструкции символическими именами, вместо последовательность нулей и единиц – машинный инструкций. Данная возможность давала разработчику сосредоточиться на процессе написания программы, а не на деталях того, как программа должна выглядеть в памяти вычислительного устройства. Процесс программирования стал более продуктивным. Язык ассемблера предоставил первую степень абстракции на пути разделения человека и машины.

Однако, языки ассемблера не являются в строгом смысле высокоуровневыми. Они не дают в распоряжение разработчика никаких дополнительных функциональных механизмов, кроме тех, которые напрямую понятны машине.

По-настоящему высокоуровневые языки стали появляются в 50-е года прошлого столетия [2]. Отличительной особенностью данных языков стали новые формы выражения алгоритма, которые напрямую не проецировались на язык машинных кодов. Требовалось некоторое промежуточное представление, которое было бы уже не понятно человеку, но еще не понятно машине.

Наличие промежуточного представления является, пожалуй, отличительной особенностью высокоуровневого языка программирования, и требует для перевода программы в целевой код полноценного транслятора.

Параллельно с появлением высокоуровневых языков программирования стали развиваться так называемые формальные языки. Точное определение того, что такое формальный язык будет дано ниже, но по сути формальный язык есть средство описания языка программирования.

Формальные языки легли в основу современной теории трансляции. Именно развитию теории формальных языков, мы обязаны множеством существующих языков программирования, первыми из которых стали COBOL, FORTRAN и, позднее, язык C [3].

1.1.1. Преимущества высокоуровневых языков

Сразу после появления первых языков программирования отличных от языка машинных кодов стало понятно, что для написания сложных программ необходим язык более дружественный человеку. Однако, парадокс заключается в том, что чем более язык дружественнее человеку, тем менее он дружественнее машине. Для написания максимально эффективных программ разработчику необходимо писать на как можно более низком уровне. Недаром, драйвера для аппаратного обеспечения компьютера даже в наши дни зачастую пишутся на языках ассемблера.

С другой стороны, по мере увеличения объема программного кода разработчик должен все выше подниматься по ступеням абстракции и должен мыслить в категориях, которые напрямую на язык машинных кодов не переводятся. Самые высокоуровневые языки позволяют разработчику полностью абстрагироваться от реализации вычислительной машины и писать программы в терминах понятных человеку.

Данное высокоуровневое описание программы требует продвинутого средства трансляции, поскольку конечная цель любой компьютерной программы – быть выполненной на электронно-вычислительном устройстве.

Перечислим основные преимуществе высокоуровневых языков по сравнению с низкоуровневыми:

1. Значительно более быстрое написание программы. Высокоуровневые языки ориентированы на человека. Они позволяют писать код в контексте поставленной задачи, а не в контексте того двоичного представления, которым является программа на самом низком уровне.

2. Более быстрая отладка программ. Высокоуровневые языки и компиляторы, их реализующие, позволяют сократить процесс отладки за счет того промежуточного представления, о котором говорилось выше. Данное промежуточное представление позволяет строить различные проекции одного и того же кода, не привязываясь к конкретной реализации.

3. Высокоуровневые языки значительно проще для чтения и сопровождения. Собственно, в этом и заключается основная цель любого высокоуровневого языка. Можно сказать, что идеальный высокоуровневый язык – это естественный язык, принятый людьми для повседневного общения. Напомним, что конечной целью естественного языка является способность выразить на нем любое понятие.

4. Высокоуровневые языки требуют значительно меньшего времен на обучение.

5. Высокоуровневые языки имеют встроенные средства для повторного использования кода. Снова, здесь можно наблюдать определенное сходство с естественными языками, которые позволяют человеку накапливать и передавать знания.

6. Высокоуровневые языки обеспечивают лучшую переносимость программ. Программу, написанную на языке ассемблера, требуется переписывать для каждой новой поддерживаемой архитектуры.

7. Транслятор позволяет встраивать в код фрагменты, необходимые для отладки программы. Другими словами, конечная программа в известном смысле пишется совместно и разработчиком, и транслятором.

1.1.2. Недостатки высокоуровневых языков

Несмотря на перечисленные достоинства, высокоуровневым языкам присущи следующие недостатки:

1. Неизбежно более низкая эффективность за счет того, что транслятор на учитывает специфические особенности целевой машины. Это может быть

связано с тем, что некоторые из таких особенностей не транслируются напрямую в те синтаксические конструкции, которые допустимы в данном высокоуровневом языке. С другой стороны, например, высокоуровневый язык С (в литературе можно встретить отнесение языка С к языкам *среднего* уровня) позволяет писать программы, которые по плотности кода и эффективности не уступают аналогичным программам, написанным на языке ассемблера.

2. Высокая сложность самого транслятора не позволяет использовать его в системах с жесткими требованиями к времени выполнения и занимаемой памяти. Что ни говори, а дополнительные издержки (*накладные расходы*) в процессе трансляции неизбежны.

Несмотря на перечисленные недостатки, современные компьютеры обладают столь впечатляющей вычислительной мощью, что можно сказать, простота и короткие сроки написания сложных программ практически всегда имеют более высокий приоритет, чем чуть большая эффективность или чуть меньшее потребление памяти.

Вместе с тем, первые высокоуровневые языки программирования, такие как FORTRAN [4], отличались низкой эффективность, главным образом из-за неэффективной трансляции. В этой связи, разработка компиляторов, генерирующих качественный код, стала одной из приоритетных проблем в программной инженерии. Данная проблема и по сей день находится в стадии активной разработки.

1.2. Реализация высокоуровневых языков

Для перевода машины с языка понятного человеку в язык понятный машине требуется отдельная программа, которую принято называть *транслятором*. Если целевой язык понятный машине – это набор инструкций, который может быть выполнен процессором, то транслятор принято называть *компилятором*. Читатель, знакомый с принципами разработки программного обеспечения, знает об исполняемых модулях с расширением EXE способных выполниться на компьютере под управлением операционной машины WINDOWS.

Помимо исполняемых модулей компилятор способе также производить библиотеки – это также набор машинных инструкций, сгруппированных в функции. Необходимо отметить, что такое понятие, как функция известно процессору, и может быть транслировано без особых проблем.

По мере того, как языки становятся более высокоуровневыми, они приобретают черты – понятия, которые напрямую процессором восприняты быть не могут. Для них нет соответствующих машинных инструкций. К

одному из таких понятий можно отнести класс – понятие из объектно-ориентированной методологии программирования [5].

Данный процесс – повышение уровня языка – требует все более сложного транслятора. В какой-то момент сложность трансляции с сохранением требований к переносимости программ между различными архитектурами становится сопоставим со сложностью написания кода. В этих случаях, если не предъявляются повышенные требования к эффективности программы, прибегают к другому методу трансляции, а именно, *интерпретации*. Классическим примером является язык Java, который не переводится напрямую в язык машинных инструкций, но переводится в некоторое промежуточное представление (байт-код), который выполняется другой специальной программой – виртуальной машиной Java (JVM). Данный подход обеспечивает лучшую переносимость программ за счет дополнительного уровня абстракции.

Интерпретация – выполнение кода не аппаратным процессором, а виртуальным – всегда менее эффективна, чем компиляция. Однако, она дает в распоряжение разработчика возможности, недоступные для компилируемых языков программирования, например, *рефлексии* – способность программы изменять себя в процессе выполнения.

Можно видеть, что процесс трансляции полон компромиссов. Чем выше должна быть целевого кода, тем более низкоуровневым должен быть исходный язык. Теория трансляции стремится к тому, чтобы на максимально высокоуровневом языке можно было бы писать максимально эффективные программы.

1.2.1. Трансляторы

По определению транслятор– это программа, переводящая текст с исходного языка на целевой язык. В первом приближении, транслятор аналогичен компилятору в тех случаях, когда целевым является язык машинных инструкций. Однако, этап синтеза, т.е. генерация целевого кода, не обязательно должен заканчиваться генерацией машинный инструкций. Транслятор должен быть спроектирован таким образом, чтобы уметь генерировать любой целевой код. В принципе, возможна трансляция, переводящая программу с одного высокоуровневого языка на другой высокоуровневый язык. Примером могут сложить первые трансляторы языка C++, которые генерировали программы на языке C, которые затем подавались на вход стандартного компилятора языка C.

1.2.2. Сложность трансляции

Разработка транслятора и языка, им реализуемого, должны вестись совместно. Необходимо тесное взаимодействие разработчика, который будет писать программы на языке, и разработчика, который проектирует транслятор. По мере увеличения сложности реализуемого языка программирования, возрастает сложность транслятора.

На первых этапах достаточно реализовать принципиальный синтаксический и семантический разбор конструкций языка и преобразование их в целевой код. Однако, этого недостаточно для того, чтобы вывести транслятор в промышленную эксплуатацию. Вторым этапом обычно является оптимизации процесса трансляции и конечного кода. На этой стадии удается добиться повышения производительности в несколько раз.

На ранних стадиях разработки инструментов трансляции, когда первые компиляторы только зарождались, не было выработано единого понимания того, что из себя должен представлять высокоуровневый язык программирования. Существовало множество отличающихся в функциональном смысле архитектур вычислительных машин и множество *рукописных* языков программирования. Здесь под словом рукописный понимается процесс создания без привлечения средств автоматизации.

В финальной стадии данного раннего этапа удалось добиться стандартизации принципиальных схем и выработать общие подходы к решению задач трансляции программы с одного языка на другой.

Данный процесс продолжается в настоящее время. Языки развиваются параллельно развитию компьютеров. Когда на рынке стали появляться первые машины с количеством вычислительных ядер более одного, компиляторам потребовалась поддержка *многопоточности*. Этот пример показателен тем, что одновременно с модернизацией языка, т.е. добавлением в него новых синтаксических и семантических конструкций, усложнялся и процесс трансляции. На практике это отразилось в том, что в процесс включались новые этапы обработки исходного кода.

Современный компилятор – это сложный программный комплекс, который не ограничивается только транслятором. Он включает в свой состав также средства сборки исполняемый модулей из набора объектных файлов и библиотек. В данный процесс также вовлечены средства отладки (*debugger* – от англ. отладчик).

Все сказанное дает представление о том, насколько сложно разработать полноценный транслятор нового языка программирования. В наше время для

того, чтобы новый язык мог стать конкурентоспособным на рынке программного обеспечения, он должен включать все ключевые свойства существующих языков и добавлять свои собственные. Разработка транслятора такого языка с нуля экономически крайне неэффективна из-за высокой сложности и длительности.

Существуют средства автоматизации процессов трансляции, с которыми мы познакомимся ниже. Автоматизация процесса трансляции основана на том, что правильно построенный компилятор должен иметь модульную структуру. К основным модулям относятся:

- лексический анализатор;
- синтаксический анализатор;
- семантический анализатор;
- генератор промежуточного кода;
- оптимизатор промежуточного кода;
- генератор целевого кода.

Перечисленные модули охватывают далеко не весь перечень средств, участвующих в процессе трансляции исходного кода в целевой. Ниже будет подробно рассматриваться каждый из перечисленных модулей.

Сейчас важно понимание основных фундаментальных принципов разработки транслятора. В частности, необходимо помнить о том, что принципиально трансляция происходит в два этапа: этап анализа исходного кода, в результате которого строится так называемое промежуточное представление, обычно в виде *синтаксического дерева разбора*. Данные этап замечателен тем, что они не учитывает никаких особенностей целевой архитектуры. Промежуточное представление позволяет удобно хранить, анализировать и оптимизировать исходный код без привязки к вычислительной машине.

Второй этап – синтез – заключается в том, чтобы перевести данное промежуточное представление в целевой код, понятный машине. Данный этап намного более творческий, поскольку необходимо учесть все особенности архитектуры процессора для того, чтобы на выходе получить максимально эффективных код.

Принципиальным здесь является то, что первый этап легко поддается автоматизации. Другими словами, существуют программные средства, позволяющие по некоторому формальному описанию языка сгенерировать для него компилятор, способный по исходному тексту программы генерировать синтаксическое дерево разбора.

В качестве простейшего примера приведем анализатор *регулярных выражений*, т.е. строк, включающих специальные синтаксические конструкции, который имеется практически в каждом современном языке программирования.

Второй этап – синтез – практически не поддается автоматизации, поскольку невозможно учесть все особенности каждой из архитектур таким образом, чтобы на выходе получить максимально эффективный код.

1.2.3. Интерпретаторы

Существует определенная тонкость, касающаяся того, что такое интерпретатор. В литературе можно встретить различные определения.

Если процесс компиляции во много стандартен и заключается в том, чтобы перевести исходный код в такой вид, который напрямую может быть воспринят процессором или виртуальной машиной, то интерпретатор работает по иному принципу.

Интерпретатор читает входную программу, записанную на исходном языке, и выполняет ее сразу, строчка за строчкой. Например, если интерпретатор встретит строку вида

```
a = b + 1,
```

он выполнит присвоение переменной *a* результата сложения значения переменной *b* с единицей. Кстати, можно видеть, как человеческое восприятие однозначно воспринимает данную строку даже без указания на то, на каком языке она записана. Формально приведенное выражение может выражать все, что угодно, например, чтение из стандартного потока ввода последовательности символов *b+1*.

Таким образом, программа исполняется не процессором, а другой специальной программой, которая в некоторой степени эмулирует работу процессора.

Интерпретация имеет несомненные преимущества, заключающиеся в том, что процесс генерации целевого кода становится в принципе не нужным. Программа исполняется непосредственно по промежуточному представлению. Однако, при этом возникают определенные проблемы. Первая и важнейшая из которых – производительность.

В случае компиляции исходный код анализируется один раз и переводится в целевой код. В случае интерпретации каждый раз для того, чтобы программа выполнилась, требуется повторный анализ кода. Отсюда

можно сделать промежуточный вывод о том, что интерпретатор в отличие от компилятора должен работать в режиме реального времени.

В действительности нет необходимости каждый раз интерпретировать код при условии, что он не изменяется. Достаточно преобразовать его в некоторую промежуточную форму, которая не требует синтаксического разбора и может быть исполнена со значительно меньшими накладными расходами и значительно большим быстродействием. В случае языка Java данной промежуточной формой является *байт-код*. В случае интерпретируемого языка PHP данной промежуточной формой являются *опкоды* (от англ. *opcode* – код операции).

Еще одной принципиальной проблемой всех интерпретаторов является необходимость использования самого интерпретатора во время выполнения каждой программы. Другими словами, помимо самой программы, занимающей определенную память, требуется в этой же самой памяти разместить программу-интерпретатор, который для любого современного интерпретируемого языка программирования является достаточно большой.

Это замечание в первую очередь относится к встраиваемым системам, для которых требования к потреблению памяти очень жесткие. Для систем с малым объемом наличной памяти применение интерпретаторов весьма ограниченно.

Подытожим все, что нам известно о том, как выполняется программа на целевой машине. Во-первых, «чистая» интерпретация, при которой код выполняется на уровне программного обеспечения с помощью специальной программы, которая анализируется исходный код и тут же его исполняет. Данный подход наиболее затратный, как с точки зрения быстродействия, так и с точки зрения потребляемой памяти. Он находит свое применение в командных строках, когда размер программы невелик, а исходный код постоянно изменяется.

Для систем, в которых исходный код меняется редко применяют второй подход, при котором текст программы анализируется один раз, переводится в некоторое промежуточное представление, которое может быть выполнено в некоторой специальной среде, например, виртуальной машине. Данный подход рекомендуется применять во всех случаях, когда эффективной программы не является главным приоритетом. Подход обеспечивает повышенную гибкость и безопасность кода.

Наконец, третий подход – компиляция – применяют в тех случаях, когда требуется высокое быстродействие, которое достигается за счет того, что

программа напрямую выполняется центральным процессорным устройством, а также низкие накладные расхода. В данном случае размер исполняемого файла практически не отличается от размера выполняемого кода и всех переменных, объявленных в программе.

Сделаем замечание относительно отношений между *программистом* и конечным *пользователем*. В случае использования интерпретируемого языка программирования конечный пользователь может даже не догадываться о том, что исходный код выполняемой программы изменился. Это возможно благодаря тому, что исходный код каждый раз разбирается заново, поэтому все изменения в нем учитываются автоматически.

С другой стороны, в случае использования компилируемого языка для того, чтобы конечный пользователь увидел изменения в программе, ее необходимо перекомпилировать из исходный кодов и заново собрать. Здесь и далее под сборкой подразумевается объединение всех объектных модулей и библиотек в один исполняемый файл.

1.3. Реализация языка программирования

Данный раздел является в некоторой степени резюме всей книги. В нем будет говориться о том, как в первом приближении реализовать некоторый язык программирования. Традиционно, целью реализации любого языка программирования является преобразование текста программы, написанного на этом языке, в представление, понятное машине.

Здесь важно уточнить, что машиной в данном случае может выступать, как реальное вычислительное устройство (компьютер), так и виртуальная машина.

Таким образом, транслятор выступает переводчиком между человеком и машиной. Перевод односторонний, т.е. от транслятора требуется умение переводить исходный код в целевой, но на наоборот. Программу, выполняющую обратную задачу, часто называют *дизассемблером*.

Очевидно, на структуру транслятора в первую очередь влияет структура языка программирования. Соответственно, сложность транслятора прямо пропорциональна сложности языка программирования. Например, сложность ассемблера значительно меньше, чем сложность компилятора языка C++.

В некоторых языках, в частности, в языке C++ стадии трансляции могут предшествовать другие стадии, например, стадия обработки текста программы *препроцессором*. Данная обработка заключается в замене всех вхождений одних строк в тексте на другие, и формально не относится к процессу компиляции.

Такие понятия, как синтаксис и семантика языка будут подробно обсуждаться ниже, однако сейчас важно упомянуть о том, что в результате трансляции синтаксис языка может измениться, а семантика – нет. Семантика включает смысловую нагрузку, которая содержится в программе, а синтаксис – описывает структуру программы. Итак, основной предпосылкой для построения транслятора является то, что при условии сохранения сценария выполнения программы, его структуру можно модифицировать.

Модификация синтаксической структуры исходного кода есть основа трансляции. Однако, модифицировать исходный код напрямую в целевой в большинстве случаев нецелесообразно, поскольку, как исходный код – понятный человеку – так и целевой код – понятный машине – не позволяют в полной мере про оптимизировать программу. Обработка исходного кода всегда осуществляется в два этапа. Первый этап – анализ, осуществляет разбор грамматической структуры программы и строит дерево разбора, которое легко анализировать и оптимизировать. Дерево разбора позволяет сохранить все семантические особенности программы, и является в некотором смысле аналогом универсального языка программирования. Его универсальность заключается в том, что по нему можно написать код для любого другого языка, при условии, что его выразительная мощность не меньше, чем мощность исходного языка.

Второй этап – синтез, генерирует целевой код по дереву разбора. Разделение структуры транслятора на эти два процесса может в значительной степени упростить его разработку.

1.3.1. Определение языка программирования

Ключевым моментом в разработке транслятора является формальное описание структуры языка программирования, для которого и создается транслятор. Описание должно в первую очередь задавать все возможные программы, которые транслятор будет считать допустимыми. Очевидно, что для любого нетривиального языка программирования количество допустимых программ будет бесконечным.

Здесь на первый план выходит разделение структуры на лексику, синтаксис и семантику. В соответствии этими понятиями ниже будет приведена принципиальная структура транслятора. Лексика, синтаксис и семантика относятся к стадии анализа исходного кода.

Структура современного языка программирования настолько сложна, что описать все возможные комбинации символов, из которых строится исходный текст программы, с помощью конечной грамматической структуры не

представляется возможным. Сложность заключается противоречии между требованиями к языку со стороны разработчика транслятора – автора языка – и конечного программиста – разработчика. Разработчик желает, чтобы язык обладал возможно большей выразительной мощью. Это желание приводит к неизбежному усложнению структуры языка. Автор языка, идя навстречу разработчику, наращивает его выразительную мощь за счет изменения его структуры. Изначальная структура языка, которую можно описать с помощью конечного набора грамматических правил, разрушается. Автору приходится или искать для языка другое описание, способное учесть все пожелания разработчика, или отказываться от тех особенностей языка, которые не могут быть описаны прежним способом. Второе приводит к снижению выразительной мощи языка и потере интереса к нему со стороны разработчиков.

Разделение процесса обработки исходного текста программы на лексический разбор, синтаксический разбор и семантический разбор приводит к значительному упрощению задачи, стоящей перед разработчиком транслятора. Для всех существующих языков программирования первые две стадии могут быть описаны формально. Творческой остается только семантический анализ, который слишком тесно связан с каждым конкретным языком и с трудом поддается формализации.

К числу основных требований к языку относятся:

- детерминированность;
- полнота;
- непротиворечивость.

Требование детерминированности означает, что каждая программа, записанная на данном языка должна интерпретироваться однозначно. Полнота языка означает, что с его помощью можно выразить любой алгоритм, который потенциально может быть выполнен на целевой машине. Непротиворечивость означает, что, если язык допускает две альтернативные конструкции, выражающие одно и то же, его трансляция в целевой код должна приводит к одному и тому же набору машинных инструкций.

Практически все современные языки программирования являются одинаковыми по своей выразительной и вычислительной мощности. Это означает, что на любую программу без учета требований к эффективности исполнения можно написать на любом языке программирования. Все языки различаются своей структурой. К сожалению, ни одна из этих структур не подходит для *формального* единообразного описания.

Таким образом, требуется *метаязык*, который бы описывал другие языки. Подобные метаязыки существуют и различаются не только средством описания, но и применимостью к той или иной стадии трансляции: лексическому разбору, синтаксическому анализу и семантическому анализу.