РОССИЙСКАЯ АКАДЕМИЯ НАУК

Сибирское отделение

Институт систем информатики

На правах рукописи

Недоря Алексей Евгеньевич

Расширяемая переносимая система программирования, основанная на биязыковом подходе

05.13.11 – математическое и программное

Обеспечение вычислительных машин и систем

Диссертация на соискание ученой степени

Кандидата физико-математических наук

Научный руководитель:

д.ф.-м.н. И.В. Поттосин

Новосибирск - 1994

С О Д Е Р Ж А Н И Е

ВВЕДЕНИЕ

В работе рассматриваются вопросы построения систем программирования (СП), удобных для разработки прикладных систем. Построение СП является основной задачей инструментального программирования. Достаточно сложным (и в большой степени субъективным) является критерий "хорошести" СП. Не претендуя на полноту определения, постараемся привести важные критерии "хорошей" СП.

1) Надежное программирование. Это свойство, очевидно, затрагивает как языки программирования, так и языковое окружение (библиотеки). Важным критерием надежности является раннее обнаружение ошибок, в первую очередь во время компиляции, и динамический контроль в тех случаях, когда статический контроль не возможен. Аккуратный выбор методов реализации окружения позволяет удалить некоторые классы ошибок полностью. Так например, встроенная в окружение сборка мусора обеспечивает отсутствие ошибок, связанных с некорректным освобождением памяти.

2) Переносимость. Нецелесообразным является разработка системы программирования для некоторой конкретной платформы (термин платформа используется как обозначение пары машина + операционная система). Изменения в аппаратуре происходят очень быстро, обостряется конкуренция среди ОС. Для подтверждения этого факта достаточно перечислить ОС, работающие на старших моделях семейства i80x86: MS-DOS, MS Windows, OS/2, Windows NT, NEXTStep, различные реализации системы Unix. Ориентация на конкретную платформу приведет к появлению неконкурентного продукта.

3) Полнота. В последнее время требования к набору библиотек СП существенно возросли, так система обязательно должна включать графическую, оконную и сетевую поддержку. К сожалению, все эти три аспекта остаются за пределами внимания разработчиков языковых окружений, так например, проект стандарта Модулы-2 не содержит библиотеки поддержки графики и окон. Список требований к СП очевидно будет расти и дальше, что приводит к следующему важному свойству "хорошей" СП.

4) Адаптируемость и расширяемость. "Хорошая" система должна предоставлять возможности развития и адаптации под различные требования. Возможность расширения (адаптации, модификации) системы – это единственный способ обеспечить выполнение новых требований, неизвестных на данный момент проектирования системы. В некотором смысле, любая программная система является расширяемой. Мы же будем называть расширяемой системой только такую систему, в которой при добавлении новых возможностей не возникает необходимость в изменении базисных понятий и механизмов.

Цель работы. Целью данной работы являлась реализация "хорошей", а именно расширяемой и переносимой системы (РПС), удовлетворяющей приведенным критериям. Работа по достижению этой цели была разбита на две подзадачи:

- выбор подходящих языков, схемы трансляции и реализация переносимых компиляторов (главы 1, 2);
- исследования принципов построения расширяемых систем и разработка системы (главы 3, 4).

Научная новизна и практическая значимость работы заключается в следующем:

1) Выделены критерии пригодности языков для реализации РПС; проведено сравнение языков с точки зрения их пригодности.

2) Предложена новая методика реализации мало-языковых транслирующих систем.

3) Реализовано семейство переносимых компиляторов и трансляторов с языков Модула-2 и Оберон-2.

4) Проведен анализ и сравнение расширяемых систем.

5) Разработаны принципы организации и структурирования РПС на базе одиночного наследования.

6) Реализовано ядро РПС Mithril.

Апробация работы. Результаты диссертации неоднократно докладывались на семинарах и конференциях, в том числе:

- 1-ая Всесоюзная школа семинар по языку Модула-2, Наманган, 1987;
- 2-ая Всесоюзная школа семинар по языку Модула-2, Севастополь, 1988;
- Рабочее совещание по архитектуре процессоров Кронос и программного обеспечения, Паланга, 1989;
- Рабочий семинар Institute of Computer Systems, ETH, Zurich, 1991;
- Международная конференция "Cooperative Standartisation", London, 1993;
- Научные семинары ВЦ СОАН СССР и ИСИ СО РАН.

Публикации. По результатам диссертации опубликовано 6 работ.

Структура и содержание работы. Работа состоит из четырех глав, заключения и двух приложений.

В первой главе дается определение расширяемой переносимой системы и определяются критерии выбора языка реализации таких систем.

Во второй главе описывается реализация переносимой компилирующей системы, являющейся необходимым инструментом создания РПС.

В третьей главе выполняется анализ расширяемых систем на примере системы Оберон. В четырех разделах анализируется структура системы и возможность ее расширения. На основании анализа перечисляются слабые места системы.

В четвертой главе описана экспериментальная РПС Mithril. При описании системы особое внимание уделяется критериям выбора проектных решений.

1. Определение РПС и выбор языка реализации

Будем называть расширяемой переносимой системой (РПС) систему, построенную по следующим принципам:

Расширяемость:

- ядро системы определяет набор базовых понятий;
- этот набор понятий может быть расширен без изменения ядра (и без перекомпиляции);
- новые понятия могут быть определены на базе уже определенных понятий;
- отсутствуют различия между базовыми и вновь определенными понятиями;

Переносимость:

- все машинно-зависимые и системно-зависимые части РПС текстуально выделены;
- объем таких частей существенно меньше объема системы;
- система может быть настроена таким образом, чтобы (почти) полностью использовать ресурсы конкретной аппаратуры и ОС.

При разработке любой системы необходимо обращать внимание на надежность системы. Для РПС критерий надежности приобретает новые черты, так как мы должны обеспечить надежное функционирование не только базовых механизмов, но и их возможных расширений и переопределений. Для обеспечения надежности в традиционных системах применяется принцип разделения системной части (супервизор) и пользовательской, при этом достаточно обеспечить надежность и независимость системной части для устойчивого функционирования системы. В расширяемых системах такого разделения не существует, поэтому особенно важным

становится использование языка программирования, обеспечивающего достаточный уровень надежности. В следующем пункте разбираются основные требования к языку реализации РПС, в том числе, и требования по надежности языка.

1.1. Основные требования к языку

Теперь рассмотрим, как из определения принципов построения системы вытекают требования к языку программирования.

Внимательный читатель может удивиться тому, что до сих пор нигде не упоминалось об объектной ориентированности (ОО). Действительно, понятие "расширяемость" сразу же наводит на мысль об использовании ОО подхода к разработке такой системы. Мы старались не использовать термин ОО по следующим причинам:

- несмотря на существование формальных моделей ОО программирования (ООП) (см. Booch [1], Meyer [2]), терминология ООП часто используется за рамками этих формальных моделей, что может привести к расхождению в понимании терминологии;
- термин ООП часто ассоциируется с программированием на языке Smalltalk [3], а в последнее время с программированием на языке C++ [4];
- вполне возможно, что существуют (или будут существовать) другие модели, более пригодные для реализации РПС.

Тем не менее, в данный момент мы не видим способа реализации РПС вне рамок ООП. В работе [5] проводится сравнение двух моделей ООП с моделью CLOS (Common Lisp Object System). В этой работе все свойства языков программирования делятся с точки зрения соответствия модели ООП на необходимые, желательные,

допустимые и недопустимые. Рассматриваются те свойства языков, которые являются существенными с точки зрения ООП, а именно: способы описания подпрограмм, отношения между объектами, свойства классов, механизм задания наследования, типизация, полиморфизм и так далее. Такой подход позволяет оценить уровень реализации одной из двух моделей ООП в некотором языке программирования, а также сравнить модели между собой. Таблица сравнения двух моделей ООП, и оценка уровня реализации этих моделей в языках CLOS и Оберон-2 приводится в приложении.

Рассмотрим некоторые свойства языков программирования в рамках моделей ООП. Будем использовать следующую нотацию:

<свойство> (B:<отношение>, M:<отношение>, <отношение>),

где B обозначает модель Буша, M - модель Мейера, а последнее отношение отражает точку зрения автора.

<отношение>:

E - Essential (необходимое)

D - Desirable (желательное)

P - Permissible (допустимое)

I - Inadmissable (недопустимое)

n/a – не определено в модели

Свойства класса:

абстракция (интерфейс отделен от реализации) (B:E,M:E,E)

инкапсуляция (B:E,M:E,E)

класс есть модуль (единица компиляции) (B:n/a,M:E,P)

класс может содержать общие переменные (т.е. доступные всем объектам класса) (B:P,M:D,D)

Отделение спецификации от реализации общепризнано является необходимым свойством языков программирования (не только ОО). Не столь общепризнано, на каком уровне происходит это отделение. Так в модели Мейера считается существенным отождествление класса и модуля. С нашей точки зрения такое отождествление приводит к слиянию двух важных структурирующих механизмов: модулей – позволяющих объединить семантически связанные компоненты, и классов, выражающих связи по наследованию. Заметим, что разделение концепций класса и модуля автоматически реализует переменные класса и методы класса, позволяет описывать в одном модуле набор тесно связанных классов, причем эта связь может быть полностью скрыта в реализации и не отражена в интерфейсах модуля и классов, тем самым заменяя механизм "друзей" в языке C++.

Подпрограммы:

могут определяться вне класса (B:D,M:P,E)

только как методы (B:P,M:D,I)

Термин "подпрограмма" здесь используется как обобщенный термин для процедуры, функции и метода. Возможность описания подпрограмм вне класса (т.е. обычных процедур) позволяет естественным образом определять методы класса без привлечения дополнительных механизмов (см. пред. пункт).

Наследование:

одиночное (B:E,M:E,E)

множественное (B:P,M:E,P)

переопределение (класс может переопределить реализацию структуры или поведения) (B:D,M:E,E)

Наследование является ключевым понятием расширяемых систем. Если необходимость одиночного наследования очевидна, то множественное наследование (МН) вызывает множественные сомнения. Часто необходимость в МН вызвана неряшливым проектированием структуры системы. Более того, легко привести пример (см. например [6]), который не реализуем при МН, но легко реализуем в терминах одиночного наследования введением дополнительного интерфейсного объекта. Вполне возможно, что существуют примеры (неизвестные автору), которые нельзя (или очень трудно) выразить в терминах одиночного наследования, поэтому мы считаем МН допустимым свойством языка. В главе 4 подробно описывается проектирование системы с использованием только одиночного наследования.

Типизация:

сильная (B:D,M:E,E), слабая (B:P,M:I,I),

статическая (B:D,M:D,E), динамическая (B:D,M:E,E).

Пожалуй, в вопросе о типизации две модели наиболее сильно отличаются. Для языка реализации РПС сильная статическая типизация является безусловным требованием. Только сильный статический контроль дает уверенность в том, что инварианты некоторой программной компоненты не будут нарушены при расширении системы. Так как не все свойства программной

компоненты могут быть проверены статически, то язык должен требовать выполнения полного набора динамических проверок. Существенным требованием к языку также является наличие динамической типизации, а именно возможности проверить, что некоторый объект является расширением данного типа. Динамическая типизация дает явный механизм идентификации типа объекта, и тем самым, существенно упрощает реализацию динамического связывания объектов, постоянных объектов (см.4.3.2) и использование гетерогенных структур данных.

Управление памятью:

сборка мусора (B:P,M:E,E).

Использование в системе сборки мусора (garbage collection, далее GC) вызывает наибольшее количество споров. Противники сборки мусора обычно выдвигают два основных аргумента:

- большие накладные расходы;
- GC не может быть использована в системах реального времени или системах с гарантированным временем отклика.

Первый аргумент не выдерживает критики, так как экспериментальные замеры показывают, что накладные расходы не превышают трех процентов. Что же касается второго аргумента, то действительно, GC не может быть использована в таких системах (вернее, использование ее требует значительных усилий), зато в большинстве других случаев GC позволяет существенно повысить надежность и полностью снять с программиста заботу о возвращении и переиспользовании памяти.

Для РПС только сборка мусора позволяет достичь приемлемого уровня надежности, так как существенно затрудняется получение информации о наличии указателей в объекте (модуль, реализующий некоторый сервис, не может знать о расширениях). Использование явного переопределения методов возвращения памяти (деструкторы C++), вместе с необходимостью отслеживать наличие ссылок на объект, приводит к необходимости реализации различных вариантов сборки мусора для различных объектов системы, и следовательно к существенному снижению надежности по сравнению с наличием одного механизма сборки мусора, встроенного в систему.

Вернемся теперь к проблеме выбора подходящего языка. Мы выработали критерии оценки ОО-модели, реализуемой языком. Но при выборе языка нам необходимо принять во внимание и более прагматические факторы, а именно:

- доступность и качество готовых реализаций;
- потенциальная эффективность языка и эффективность реализаций;
- объем требуемых ресурсов;
- трудоемкость реализации.

Необходимость статической типизации позволила нам существенно уменьшить число рассматриваемых языков. Кроме того, мы не стали рассматривать такие языки, как Eiffel и Sather, которые видимо имеют большое будущее, но в настоящее время мало доступны.

1.2. Сравнение языков Модула-3, Оберон, Оберон-2, C++

В данном разделе мы сравним языки семейства Модулы-2 [7], а именно языки Модула-3 [8], Оберон [9], Оберон-2 [10] и язык C++.

1.2.1. Народ vs. C++

Язык C++ не подходит в качестве языка разработки РПС, так как в нем не реализованы многие существенные свойства таких языков:

- разделение концепции модуля и класса;
- сильная статическая типизация;
- динамическая типизация;
- сборка мусора.

Множественное наследование, реализованное в версиях языка, начиная с 2.0, мы также склонны считать скорее недостатком языка, так как использование этого механизма часто приводит к построению плохо структурированной системы.

1.2.2. Модула-3 vs. Оберон

Разработчики языка Модула-3 пытались объединить лучшие черты языков Mesa, Модула-2, Cedar и Модула-2+ [11]. Полученный язык действительно весьма привлекателен, но, к сожалению, страдает некоторым гигантизмом. Помимо поддержки ООП, Модула-3 поддерживает исключения, финализацию, параллелизм и весьма сложный механизм раздельной компиляции, требующий этапа связывания всей программы, что является абсолютно недопустимым для РПС. Для реализации языка требуется большая динамическая поддержка, что затрудняет реализацию для машин с ограниченными ресурсами. Механизм ассоциирования методов с объектами позволяет реализовать как методы, ассоциированные с классом, так и методы, ассоциированные с экземпляром. В то же время в языке отсутствуют операции проверки динамического типа объекта, что является необходимым требованием для нашей модели ООП. Язык содержит

набор низкоуровневых возможностей, но, в отличие от языка Модула-2, использование этих возможностей разрешено только в специальных низкоуровневых (unsafe) модулях.

Язык Оберон, напротив, самый маленький язык семейства языков, ведущих свой род от Алгола-60. При разработке языка было удалено множество конструкций языка Модула-2 и добавлено только расширение типа и сборка мусора. Язык исключительно прост в реализации (естественно, кроме сборки мусора) и позволяет разрабатывать эффективные ОО программы.

Помимо очевидных преимуществ, связанных с размером языка (и динамической поддержкой), на наше решение повлияло то, что несмотря на название, язык Оберон гораздо ближе по духу к языку Модула-2, чем Модула-3.

1.2.3. Оберон vs. Оберон-2

Противопоставление этих языков невозможно, так как Оберон-2 является строгим расширением языка Оберон. Оба языка удовлетворяют всем необходимым требованием модели ООП. Поэтому просто перечислим дополнительные возможности языка Оберон-2:

– методы (type-bound procedure) позволяют аккуратно разделить операции над каждым объектом и переопределять одни операции, не изменяя других; повышают эффективность, ликвидируя лишние динамические проверки типов; повышают надежность ввиду неявной инициализации методов;

– экспорт только для чтения (read only export) позволяет повысить эффективность доступа к атрибутам объектов (без ущерба для надежности);

– динамические массивы позволяют повысить универсальность операций и избавиться от ограничений в системе.

В то же время новые возможности не усложняют реализацию языка. Любой текст на языке Оберон является также текстом на языке Оберон-2.

1.2.4. Выводы

Итак, как и следовало ожидать, проведенный анализ подтвердил правильность выбора языка Оберон-2 для реализации РПС. Некоторой проблемой является отсутствие низкоуровневых средств в этом языке. Мы считаем это скорее сильной чертой языка, чем недостатком. Язык программирования высокого уровня не должен содержать "опасных" средств. Возникает вопрос о языках программирования таких низкоуровневых частей системы, как динамическая поддержка, сборка мусора, реакция на программные прерывания и т.д. Эти части системы не являются переносимыми, и могут быть написаны на разных языках для различных платформ. Но мы считаем, что трудоемкость переноса может быть существенно уменьшена, если система содержит еще и язык программирования низкого уровня. В качестве такого языка мы используем язык Модула-2. Может показаться странным использование Модулы-2 в качестве языка низкого уровня. Но заметим, что Модула-2 обладает очень важным свойством, присущим скорее языку низкого уровня, чем ЯВУ: отсутствие собственной динамической поддержки.

Язык Модула-2 обладает многими важными особенностями, упрощающими реализацию на нем динамической поддержки:

- раздельная компиляция позволяет выделить набор переносимых библиотек и отделить их от тех частей системы, которые обязательно должны быть переписаны при переносе;

- средства низкого уровня позволяют достаточно удобно выполнять низкоуровневые действия;

- наличие механизма кодовых процедур позволяет достигнуть максимальной эффективности;

- алгоритмы, не требующие использования средств низкого уровня, могут быть написаны с надежностью, характерной для ЯВУ.

И еще одно свойство языка Модула-2 является очень важным для нас: близость его к языку Оберон-2. Это позволяет легко использовать Модула-2 библиотеки при программировании на Обероне и тем самым не потерять весь объем программного обеспечения, накопленного нами на языке Модула-2.

Мы считаем, что наличие пары языков в системе, близких по синтаксису и семантике, но отличающихся некоторыми существенными свойствами (Оберон-2: расширение типа, сборка мусора; Модула-2: низкоуровневое программирование, отсутствие динамической поддержки) позволяет значительно увеличить общий потенциал системы программирования.

2. Семейство Modula-2/Oberon-2 компиляторов и трансляторов

Согласно анализу, проведенному в главе 1, для хорошей жизни нам нужны реализации языков Оберон-2 и Модула-2, причем речь не идет о реализации этих языков для какой-либо конкретной машины, так как наша цель - разработка переносимой системы.

Компиляторы с языка Модула-2 реализованы на всех известных нам машинах, и основная проблема - это отсутствие стандарта и проблема совместимости различных версий. Заметим, что реализация языка на станциях КРОНОС содержит набор расширений языка и так же не полностью совместима с другими реализациями.

Еще более сложна проблема с реализацией языка Оберон-2. В момент начала проекта (весна 1991 г.) таких реализаций просто не существовало, а язык Оберон был реализован только на машинах, нам не недоступных (Ceres, Mac, SparcStation).

Единственным выходом была разработка собственного переносимого компилятора с языка Оберон-2. Забегая немного вперед, заметим, что наша реализация языка Оберон-2 появилась практически одновременно с появлением окончательной версии языка. Это стало возможно благодаря постоянной связи с одним из авторов языка проф. Моссенбоком.

Реализация языка Модула-2/КРОНОС не выдерживала критики с точки зрения переносимости. Поэтому было принято решение о разработке пары переносимых компиляторов.

В дальнейшем изложении мы будем использовать термин "компилятор" в тех случаях, когда речь идет о любом из этих двух компиляторов.

2.1. Выбор схемы трансляции и понятие биязыковой транслирующей системы

Рассмотрим способы реализации пары переносимых компиляторов. Эти компиляторы можно было бы реализовать полностью независимо, но хотелось найти способы, позволяющие уменьшить затраты на разработку. Очевидное решение - это реализовать многоязыковую транслирующую систему, позволяющую для N языков и для M целевых машин реализовать N фаз анализа, транслирующих некоторой язык программирования в промежуточное представление, и M фаз синтеза, транслирующих промежуточное представление в коды конкретной ЭВМ, и тем самым существенно сократить затраты на реализацию (с N*M до N+M).

Такой общий подход принят в системе Бета [12]. Промежуточное представление определяется языком ВЯЗ. В рамках проекта были выполнены реализации языков Паскаль, Симула-67 и Фортран и подмножества языков Модула-2 и Ада. Наибольшая нагрузка в проекте выпадает на внутреннее представление (внутренний язык). Требования, предъявляемые к ВЯЗу, весьма противоречивы. С одной стороны, все конструкции языков программирования должны достаточно удобно отображаться в него, с другой стороны, он должен быть достаточно удобен для генерации кода. Соответственно, уровень его должен быть достаточно низок (очевидно, что все операторы управления могут быть представлены в терминах переходов управления), и достаточно высок, для того чтобы можно было выполнять оптимизацию и генерацию кода на разные машины.

Внутренний язык построен как объединение конструкций входных языков системы. Попытки совместить несовместимое привели к большому объему как внутреннего языка, так и системы в целом.

Отрицательный результат экспериментов с многоязыковыми системами (не только с системой Бета) является отражением того факта, что проблема в целом неразрешима (или пока неразрешима) и для достижения успеха необходимо пойти на некоторые ограничения. Если для систем типа системы Бета использовать обозначение N->M, то естественно рассмотреть случаи, когда N или M равно 1, то есть компиляция одного языка на множество машин, или компиляция множества языков на одну машину. Введение таких ограничений оказывается вполне достаточным. Примером удачной системы N->1 является система TopSpeed, в которой реализованы языки Модула-2, Паскаль, C, C++ для IBM PC, а примером удачной системы 1->M является Оберон-компилятор OP2 [13], который в настоящее время перенесен на такие существенно различные архитектуры, как SPARC [14], MC 680x0 [15], MIPS, RS/6000 и i80386.

Как нам кажется, успех таких систем обусловлен тем, что в обоих случаях существует определенность на одном конце системы. Для системы 1->M определен входной язык, и вся дальнейшая работа может выполняться в его терминах. Промежуточное представление соответствует входному языку. Для системы N->1 задана целевая архитектура, и каждый входной язык системы транслируется в некоторое промежуточное представление, близкое к целевой архитектуре. Необходимо также отметить неявное задание семантики, что существенно облегчает разработку промежуточного представления. В случае неограниченной системы (N->M) все семантические особенности языков должны быть явно отражены в промежуточном представлении.

Проведенный анализ показывает трудности разработки многоязыковых систем. С другой стороны, мы не можем ограничиться схемой 1->M или N->1. Выход из этого тупика нам дает анализ необходимых для системы входных языков. Язык Оберон-2 является расширенным подмножеством языка Модула-2. Большинство конструкций этих языков обладают сходной семантикой.

Таким образом, мы приходим к отображению 1+e -> M, где e определяет количество дополнительных конструкций, необходимых для реализации второго входного языка (e << 1). При таком подходе мы будем (как и в Бете) строить промежуточное представление как отображение объединения двух языков, но, в отличие от системы Бета, набор входных языков определен изначально и не может быть расширен, и, что очень важно, семантика этих языков для большинства конструкций совпадает. Так как промежуточное представление является общим для обоих языков, то и фазы синтеза для каждой целевой архитектуры совпадают, а при реализации фазы анализа мы выделили все общие компоненты компилятора (подробнее см. 2.4).

Следующий важный вопрос - это выбор интерфейса между фазами анализа и синтеза. Можно представить следующие варианты такого интерфейса: процедурный интерфейс, промежуточный язык и промежуточная структура в памяти.

Процедурный интерфейс

Интерфейс между фазой анализа и синтеза оформляется в виде набора процедур. Анализ вызывает процедуру, соответствующую синтаксической (под-)конструкции после разбора данной конструкции. Так, разбор условного оператора может быть задан

процедурами if, else, endif. Например, при компиляции условного оператора

```
  IF expr1 THEN statseq1
  ELSIF expr2 THEN statseq2
  ELSE statseq3
  END;
```

может быть порождена последовательность вызовов:

```
  if(expr1); (* statseq1 *)
  else();
    if(expr2); (* statseq2 *)
    else() (* statseq2 *)
    endif;
  endif;
```

С первого взгляда такой интерфейс выглядит весьма привлекательным. Простой неоптимизирующий генератор может немедленно порождать код, а оптимизирующий генератор может строить некоторую удобную структуру для дальнейшей генерации. В реальной жизни немедленно возникают проблемы, связанные с тем, что для генерации хорошего кода необходим анализ достаточно большого фрагмента текста (например, для распределения регистров). Таким образом генерация вынуждена (почти) всегда восстанавливать из вызовов структуру текста. Причем построение такой структуры должно выполняться в каждом генераторе кода. Задача построения промежуточного представления более просто и естественно может быть выполнена на фазе анализа, так как фаза анализа обладает полной информацией о структуре единицы компиляции.

Процедурный интерфейс был реализован автором в компиляторе mx [16]. Кроме генератора для ЭВМ КРОНОС [17,18], была также реализована версия генерации для NS32x32. Эти эксперименты показали, что реализация переносимого компилятора по этой схеме хотя и возможна, но требует больших усилий.

Промежуточный язык

При этом подходе фаза анализа строит текст на некотором промежуточном языке, а фаза синтеза выполняет разбор текста на промежуточном языке, а затем выполняет генерацию кода. В практике разработки компиляторов промежуточные языки в основном использовались не для построения переносимых компиляторов, а для разбиения компилятора на последовательность достаточно небольших проходов, которые могли выполняться на ограниченных ресурсах системы. Блестящим примером такого подхода является шестипроходный Concurrent Pascal компилятор [19] для ЭВМ PDP-11. Другим примером является четырехпроходный Модула-2 компилятор, число проходов которого изначально было определено объемом памяти ЭВМ Lilith, и который затем был перенесен на другие платформы. Для этого необходимо было переписать проходы фазы синтеза. Промежуточные языки также широко используются в многоязыковых транслирующих системах [12].

Недостатки этого подхода являются непрерывным продолжением его достоинств: фаза синтеза должна повторять (упрощенный) анализ, значительное время тратится на запись и чтение текстов на промежуточном языке. Как и в предыдущем варианте построение структуры, пригодной для генерации кода, должно выполняться в каждом генераторе.

Промежуточная структура в памяти

При этом подходе фаза анализа строит в памяти некоторую промежуточную структуру, а фаза синтеза обходит ее при генерации кода. Эта схема стала использоваться сравнительно недавно, с того момента, как объем оперативной памяти стал достаточно большим для хранения такой структуры. Такой подход неприменим для систем с ограниченными ресурсами памяти. В дальнейшем мы покажем, какие объемы памяти достаточны для его использования.

Основной проблемой является выбор промежуточной структуры. Так как структура строится машинно-независимой фазой анализа, то она должна быть достаточно удобна для всех генераторов кода. Одним из возможных подходов является выбор представления, удобного для генерации кода для некоторой идеальной или усредненной архитектуры. К сожалению, такой выбор приводит к тому, что генерация кода становится неудобной для любой конкретной архитектуры.

Другой вариант - это привязка промежуточной структуры к языку программирования, то есть, например, промежуточная структура есть синтаксическое дерево единицы компиляции. Этот подход нам кажется наиболее подходящим для наших целей. Он не позволяет строить многоязыковые системы, но зато фаза анализа действительно становится машинно-независимой, так как выходом ее является машинно-независимое синтаксическое дерево. Необходимо заметить, что некоторая зависимость этого дерева от целевой машины/системы все-таки существует. Подробнее об этом будет сказано в пункте 2.3.

И последний оставшийся вопрос - выбор языка реализации. Очевидно, что для облегчения переноса сам компилятор должен быть реализован на одном из двух входных языков системы. Мы полагаем, что как язык реализации компиляторов, Оберон-2 не имеет особых преимуществ перед Модулой-2. Использование объектно-ориентированной техники кажется не полностью оправданным для компилятора, так как компилятор – это законченный программный продукт, не требующий расширений.

2.2. Входные языки системы

Существенно разные решения были приняты при фиксации входных языков системы. Мы полагаем, для языка Оберон-2 – основного входного языка системы - важно реализовать стандарт языка [10]. Поэтому мы добавили только несколько расширений, не изменяя языка.

Ситуация с языком Модула-2 является в данный момент достаточно неясной. Во-первых, существует стандарт де-факто языка PIM3 [7] и проект стандарта языка, который в данный момент готовит рабочая группа WG13 ISO. Некоторые черты языка существенно отличаются в этих двух описаниях. Во-вторых, каждая известная нам реализация языка вносила существенные изменения. Так, достаточно далекой как от PIM3, так и от проекта стандарта является реализация языка в системе TopSpeed.

Мы попытались найти некоторое равновесие между проектом стандарта, PIM3 и, кроме того, приблизить язык к языку Оберон-2. Кратко перечислим основные особенности реализованной версии языка Модула-2.

1) Базовые типы

Набор числовых типов существенно расширен; кроме типов INTEGER, CARDINAL, REAL и LONGREAL добавлены типы SHORTINT, LONGINT, SHORTCARD, LONGCARD. Важной особенностью реализации является понятие включения типов (см. Оберон), то есть значение некоторого числового типа является также значением объемлющего числового типа. Компилятор поддерживает следующую иерархию типов:

```
SHORTINT <= INTEGER <= LONGINT
                                    <= REAL <= LONGREAL
SHORTCARD <= CARDINAL <= LONGCARD
```

Знак "<=" в данном случае обозначает включение.

2) Многомерные динамические массивы

В соответствии с проектом стандарта, параметр процедуры может быть многомерным открытым массивом. Также реализованы многомерные динамические массивы в духе Оберона-2.

Например:

```
TYPE
  Matrix = ARRAY OF ARRAY OF REAL; -- открытый двумерный массив
  pMatrix = POINTER TO Matrix; -- динамический двумерный массив

PROCEDURE add(VAR res: Matrix; a,b: Matrix);
     ...
```

```
END add;

VAR x: POINTER TO Matrix;
...
  NEW(x,10,20); -- отведение памяти под массив 10x20
```

3) Экспорт только для чтения

Переменные в определяющем модуле могут быть помечены символом "-":

```
VAR root-: NODE;
```

При этом изменение значения таких переменных возможно только в процедурах данного модуля.

4) Переименование при импорте

В секции импорта можно задать внутреннее имя для импортируемого модуля. Это свойство взято из языка Оберон.

5) Процедуры с переменным числом параметров

Введен специальный синтаксис для процедур с переменным числом параметров. Последний параметр такой процедуры описывается в виде:

```
SEQ ident: BYTE
```

Вместо такого параметра может быть подставлена любая последовательность параметров (в том числе и пустая) любого

типа. Это низкоуровневое средство позволяет реализовать процедуры форматного вывода, аналогичные стандартной процедуре printf библиотек языка C.

```
PROCEDURE printf(format: ARRAY OF CHAR; SEQ args: BYTE);
```

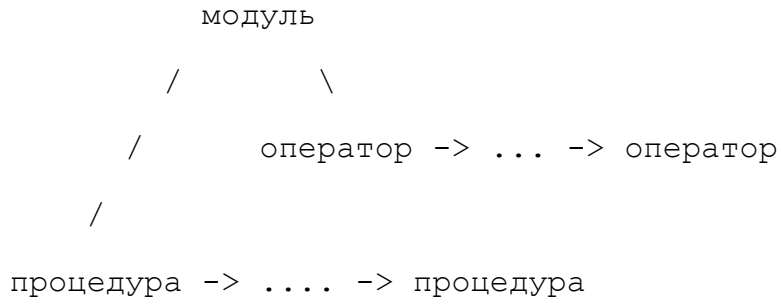
Это расширение упрощает реализацию библиотек ввода/вывода, которые являются узким местом во всех реализациях Модулы-2.

2.3. Внутреннее представление

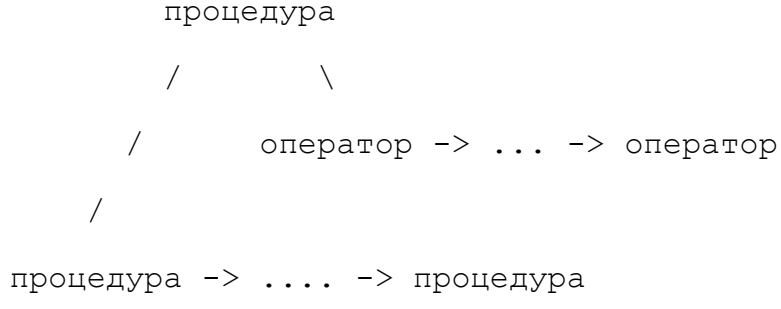
Как уже упоминалось в пункте 2.2, для внутреннего представления нами выбрано синтаксическое дерево (СД) единицы компиляции. Это представление не является идеальным для выполнения оптимизаций и качественной генерации. Впрочем, для разных оптимизаций требуется различная форма представления. Мы полагаем, что для выполнения оптимизаций дерево может перестраиваться в некоторую удобную форму. Такое преобразование гораздо удобней выполнять с деревом, чем с другими видами промежуточных представлений. В двух реализованных генерациях (см. 2.6.1, 2.6.3) для оптимизации структур управления по дереву строится граф управления. Для простой, неоптимизирующей генерации СД достаточно удобно.

Большое влияние на выбор формы СД оказало внутреннее представление Оберон-2 компилятора OP2, описанное в [13]. Мы взяли это представление за основу и добавили конструкции, соответствующие расширенному набору типов и операторов языка Модула-2.

Внутреннее представление образовано узлами четырех видов: NODE (Узел), OBJECT (Объект), STRUCT (Структура) и VALUE (Значение). Операторная часть программы отображается в дерево, состоящее из Узлов. Кроме того, в дерево входят Узлы, соответствующие единице компиляции и процедурам. Кроме ссылок на левое и правое поддеревья Узел содержит ссылку для определения последовательности Узлов. Единица компиляции отображается в дерево вида:



В свою очередь,



Блок (процедура или модуль) содержит ссылку на последовательность операторов и на список вложенных процедур. Локальные модули не отображаются в синтаксическом дереве.

Узел, соответствующий объекту (переменной, константе или процедуре), содержит ссылку на описатель объекта (запись типа OBJECT). Все узлы, кроме операторных, содержат ссылку на типовое значение (запись вида STRUCT). Объекты-константы и узлы,

соответствующие константам и литералам, содержат ссылку на значение (запись типа VALUE).

Приведем примеры представления некоторых конструкций.

Для описания типа:

```
TYPE T = POINTER TO ARRAY OF CHAR;
```

будет построен объект с видом "тип" и два типовых значения с видами "указатель" и "открытый массив"

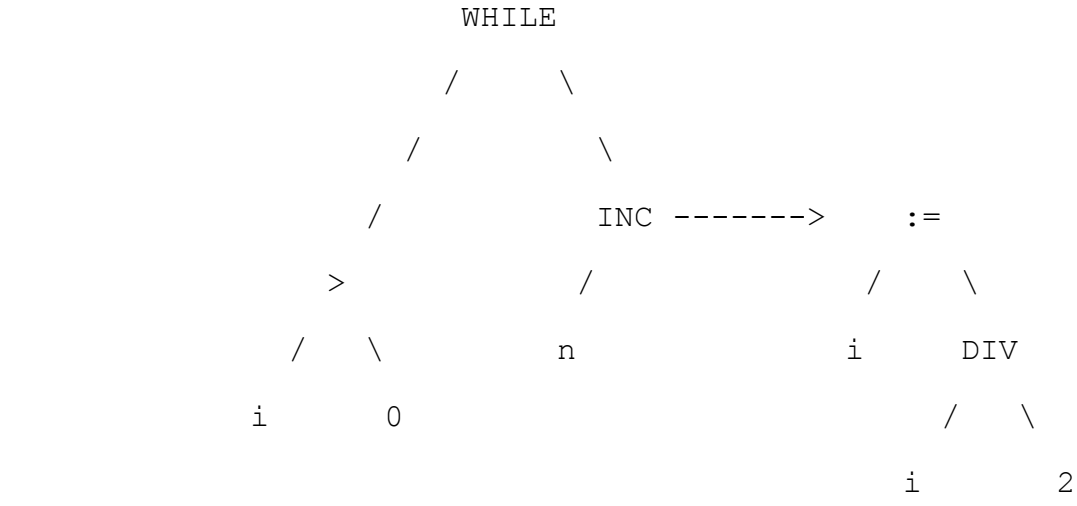
```
        OBJECT      STRUCT      STRUCT      STRUCT


         type
         base ----> pointer
         "T"        base   ---> array_of
                                 base    ----> char
```

Оператор

```
        WHILE i>0 DO INC(n); i:=i DIV 2 END;
```

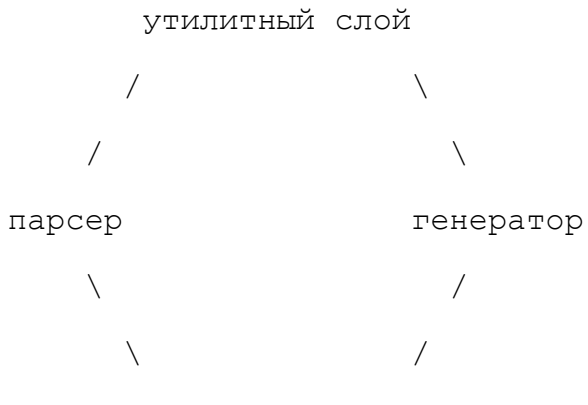
будет отображен в следующую структуру



Важно отметить, что в СД отображаются все описания и операторы обоих входных языков. Это не только уменьшает затраты на перенос компилятора, но и позволяет писать на смеси языков. Так, в Оберон-2 модуле можно использовать типы, которых нет в этом языке (например, SET OF). Для этого достаточно проимпортировать тип из модуля на языке Модула-2.

2.4. Структура компилятора

Компилятор состоит из интерфейсного слоя, парсера (front-end), генератора (back-end) и утилитного слоя.



Интерфейсный слой содержит операции взаимодействия с системой, утилитный слой определяет операции чтения исходного текста и выдачу сообщений компилятора. Меняя утилитный слой, мы можем перейти от обычного пакетного компилятора к турбо-компилятору, читающему текст из буфера редактора и так далее.

Алгоритм компиляции выглядит следующим образом (cu – корень синтаксического дерева единицы компиляции):

```
инициализация
cu:=Парсер.построение_синтаксического_дерева();
Генератор.размещение_объектов_модуля(cu);
```

```
IF определяющий-модуль или оберон-модуль THEN
  Парсер.запись_симфайла
END;
IF реализующий или программный-модуль или оберон-модуль THEN
  Генератор.генерация_кода(cu)
END
```

Парсер выполняет синтаксический и семантический анализ, строит СД единицы компиляции и возвращает его корень. Если при этом были обнаружены ошибки, то компиляции завершается. Генератор всегда работает с корректным синтаксическим деревом. Генератор выполняет проход по объектам модуля, вычисляя необходимые атрибуты: размер типов, размещение переменных, размер области параметров и т.д., после чего выполняется запись симфайла. Затем выполняется собственно генерация кода.

Вычисление атрибутов объектов перед записью симфайла необходимо, так как в симфайле должна присутствовать информация о них. Так, для генерации доступа к переменной из другого модуля необходимо знать размещение этой переменной.

Структура генератора для разных платформ может сильно различаться (см. 2.6). Генератор может выполняться в несколько проходов по СД и может включать в себя фазу машинно-зависимой оптимизации. Машинно-независимая оптимизация может быть вставлена между фазами анализа и синтеза.

Рассмотрим подробнее структуру парсера. Парсер состоит из ядра, реализующего следующие функции:

лексический анализ;

описание и поиск объектов, чтение/запись симфайлов;

построение СД, семантический контроль.

Это ядро является общим для входных языков. Кроме этого, для каждого языка существует модуль, реализующий синтаксический анализ. Для того чтобы перейти от Модула-2 компилятора к Оберон-2 компилятору, достаточно сменить только один модуль.

Такая реализация парсера возможна только благодаря близости входных языков. При разработке парсера мы старались выбирать алгоритмы, учитывающие разницу языков естественным образом. В качестве примера рассмотрим алгоритм идентификации. С точки зрения идентификации разница в языках заключается в следующем:

- локальные модули есть только в Модуле-2;
- расширение типа записи есть только в Обероне-2.

Для реализации общего механизма нужно считать, что все записи в Модуле-2 есть базовые записи. После этого никакого изменения или настройки алгоритма идентификации не требуется при переходе от одного языка к другому.

В некоторых случаях все же приходится выполнять некоторую настройку на входной язык. Так, например, при лексическом анализе необходимо изменить множество ключевых слов. Число таких мест в компиляторе весьма невелико.

Компилятор взаимодействует с системой только через модули интерфейсного и утилитного слоя. Интерфейсный слой состоит из модуля, реализующего функции, необходимые парсеру: управление памятью, чтение/запись симфайла, сервисные и отладочные функции. Как правило, для реализации генератора необходим модуль интерфейсного слоя, реализующий дополнительные функции взаимодействия с системой (например, запись объектного файла).

Три (определяемые пользователем) файла используются интерфейсным слоем для построения удобной среды

программирования: файл конфигурации, файл поиска и файл сообщений.

Файл конфигурации задает различные параметры, такие как расширения файлов, ограничения на число ошибок и т.д..

Файл поиска (redirection file) задает пути поиска файлов и директории для записи создаваемых файлов.

Файл сообщений содержит тексты сообщений об ошибках и тем самым позволяет переводить их на любой подходящий язык, будь то английский, русский или исконно русский.

Утилитный слой определяет способ доступа к исходному тексту и способ выдачи сообщений об ошибках. Для инструментальной машины характерно большое количество вариантов утилитного слоя и, тем самым, различных компиляторов. Так, в инструментальной среде на Кроносе в данное время используется 12 = 2*2*3 различных компиляторов:

два входных языка

пакетный компилятор/турбо-компилятор

генерация для Кроноса/трансляция в ANSI C/генерация для i386.

Модуль интерфейсного слоя выполняет еще одну важную функцию - определение параметров окружения. Эти параметры задают ограничения, необходимые для корректной работы фазы анализа. Часть параметров задают лексические ограничения, например, число цифр в мантиссе и максимальную экспоненту. Задаются также диапазоны представления базовых типов, необходимые как лексическому анализу, так и для проверки корректности выполнения константных выражений. Такая параметризация позволяет выполнять настройку парсера на особенности конкретной архитектуры. Меняя

параметры мы можем определить параметры целевой архитектуры и получить тем самым кросс-компилятор.

Остается заметить, что синтаксический анализ выполняется методом рекурсивного спуска, а таблица символов реализована на базе бинарных деревьев.

2.5. Взаимодействие генератора с парсером

В компиляторах семейства принята довольно сложная модель взаимодействия машинно-зависимой части (генератора) с машинно-независимой. Эта модель позволяет сохранять парсер неизменным при переносе на новую платформу.

Рассмотрим различные способы взаимодействия. Следующая схема определяет порядок выполнения частей парсера и генератора. Здесь на одной строке записаны действия, выполняемые на одной стадии.

Парсер	Генератор
чтение симфайлов	определение параметров окружения чтение машинно-зависимой части симфайлов
синтаксический и семантический анализ	анализ машинно-зависимых операций
запись симфайла	размещение объектов запись машинно-зависимой части симфайла обход дерева и генерация кода

На этой схеме видно, что некоторые операции, определяемые генератором, вызываются парсером при чтении/записи симфайлов и разборе текста.

Общая структура симфайла является машинно-независимой и определяется парсером. Симфайл содержит информацию об экспортируемых объектах и типах, необходимых для выполнения семантического анализа. В то же время в симфайле должна содержаться машинно-зависимая информация, необходимая для генерации кода при обращении к внешним объектам. Таким образом, симфайл должен содержать как машинно-независимую информацию (достаточно сложно устроенную, см. [20,21,22]), так и машинно-зависимую информацию. Важно заметить, что объем и вид машинно-зависимой информации сильно зависит от генератора.

При разработке компилятора рассматривалось несколько способов реализации взаимодействия при чтении/записи симфайлов. Одним из вариантов было полное исключение машинно-зависимой информации из симфайла и перевычисление ее каждый раз при использовании. Очевидным недостатком этого подхода является некоторая потеря эффективности. Но не это, а более веские причины заставили отказаться от этого варианта:

- вообще говоря, в этом случае симфайл должен также содержать для каждого объекта множество опций компилятора, так как от этого может зависеть размещение объекта;

- потеря преемственности: при модификации алгоритма вычисления размещения объекта может нарушиться согласованность размещений при описании и использовании объекта.

Этот вариант был отброшен, и рассматривались только различные способы размещения информации. Был выбран наиболее простой способ: после записи машинно-зависимой информации об

объекте вызывается процедура генерации, которая записывает дополнительные атрибуты (аналогично - при чтении объекта).

Точно такой же способ используется для выполнения анализа машинно-зависимых операций (например, операции преобразования типа). Парсер строит (под-)дерево и вызывает процедуру генератора, контролирующую корректность преобразования для данной архитектуры.

Так как прямое обращение (то есть импорт) из парсера к процедурам генератора невозможно, то парсер параметризован процедурами чтения/записи машинно-зависимых атрибутов, вычисления машинно-зависимых выражений и, кроме того, множествами, определяющими совместимость по присваиванию для объектов машинно-зависимымых типов BYTE и WORD.

Таким образом для реализации генератора необходимо выполнить следующие действия:

- определить параметры окружения;
- реализовать операцию вычисления машинно-зависимых операций;
- реализовать обход всех объектов единицы компиляции для вычисления машинно-зависимых атрибутов;
- реализовать операции чтения/записи машинно-зависимых атрибутов;
- реализовать генерацию кода (маленький такой пунктик).

2.6. Состав семейства

В настоящее время в состав семейства входят компиляторы для рабочих станций Кронос и для машин на базе i386/486 и

трансляторы в ANSI C. В следующих пунктах дается краткое описание особенностей разработки генерации.

2.6.1. Компиляторы для рабочих станций Кронос

Генератор кода для Кроноса был переделан из генерации компилятора mx. Тем самым мы получили возможность сравнить варианты промежуточного представления (СД и процедурный интерфейс) с точки зрения удобства генерации. Этот переход позволил существенно упростить алгоритмы генерации кода и улучшить качество кода. Хотя мы и не ставили целью добиться высокого качества кода, так как мы рассматриваем Кронос только как инструментальную машину. Генератор выполняет щелевую оптимизацию (уменьшая количество доступов к памяти) и оптимизацию структур управления. Для оптимизации структуры управления по СД строится граф управления, узлами которого являются линейные участки и переходы. При оптимизации из этого графа удаляются недоступные части, выполняется сливание перехода на переход и некоторые другие преобразования.

Компилятор является стандартным с точки зрения ОС Excelsior, то есть он порождает стандартный кодофайл и стандартный файл с отладочной информацией. Компилятор позволяет импортировать модули, скомпилированные компилятором mx. Для этого реализована специальная утилита "конвертор симфайлов", которая переводит симфайл из стандарта mx в стандарт переносимого компилятора.

Оберон-2 компилятор дополнительно записывает в кодофайл образ типовой системы. Этот образ необходим для реализации операций управления памятью, сборки мусора и динамической

типизации. Модуль динамической поддержки (RTS) неявно импортируется в каждый Оберон-модуль. При инициализации модуля RTS по образу типовой системы строит структуры, необходимые для его дальнейшей работы. Описание и анализ таких структур можно найти в работе [23]. Модуль динамической поддержки реализован А.Хапугиным.

2.6.2. Трансляторы в ANSI C

В качестве одного из генераторов семейства нами реализован генератор, порождающий текст на языке ANSI C. В последнее время язык C часто используется в качестве промежуточного языка. Так, большинство компиляторов с современных ОО языков (Eiffel, Sather, Modula-3) порождают текст на языке C. В первую очередь это объясняется широким распространением языка C. Реализовав генерацию в C, мы фактически получаем компилятор для всех машин сразу. С другой стороны, язык C достаточно удобен в качестве промежуточного языка. Мы рассматриваем язык C в качестве переносимого ассемблера, неудобного и ненадежного для ручного программирования, но вполне подходящего для генерации.

Язык C не создавался для использования в качестве промежуточного языка, и он обладает некоторыми недостатками, усложняющими трансляцию на него:

- отсутствие вложенных процедур;
- отсутствие возможности отслеживания переполнений;
- трудности в реализации переносимой сборки мусора;
- различные ограничения в разных реализациях.

Транслятор в ANSI C может быть использован двумя разными способами. В первом варианте такой транслятор является первым проходом компилятора, причем вторым проходом вызывается C компилятор. Во втором варианте используется собственно порожденный текст на языке C; например, он переносится на другую платформу.

Два разных способа использования транслятора предъявляют разные требования к порожденному коду. В первом случае важна его оптимальность, во втором случае адекватность тексту на исходном языке, так как дальнейшая отладка выполняется в терминах языка C. Мы уделяли большее внимание второму варианту использования.

Генератор состоит из головного модуля, реализующего процедуры

"размещение_объектов_модуля" и

"генерация_кода" (см. 2.4)

и компонент, реализующих следующие функции:

генерация описаний;

генерация выражений;

генерация операторов.

Кроме того, генератор использует дополнительный модуль интерфейсного слоя, реализующий элементарные операции записи текста. Все модули, порожденные транслятором, импортируют библиотеку, в которой определены базовые типы и реализован набор вспомогательных функций и функций динамической поддержки.

Модуль генераций описаний порождает описания объектов на языке C по их внутреннему представлению. Практически всем типам и объектам языков Оберон-2/Модула-2 соответствуют аналоги в

языке C. Исключение составляют вариантные записи и расширенные записи.

Приведем примеры трансляции описаний. Мы будем опускать некоторые детали для упрощения примеров.

Modula-2	C
TYPE P = POINTER TO INTEGER;	typedef INTEGER *P;

TYPE
  R = RECORD
  CASE male: BOOLEAN OF
  |TRUE : salary: REAL;
  |FALSE: beauty: LONGINT;
  END;
  END;
	typedef struct R_STR {
	  BOOLEAN male;
	  union {
	  REAL salary;
	  LONGINT beauty;
	  } X1;
	} R;

TYPE
  Vector = POINTER TO
  ARRAY OF REAL;
 	
  Matrix = ARRAY [0..7] OF Vector;
	typedef struct
	  { INDEX Len0; } *Vector;
	
	typedef Vector Matrix[8];

VAR
  matrix: Matrix;
  i,j: INTEGER;
  r: REAL;
	Matrix matrix;
	INTEGER i;
	INTEGER j;
	REAL r;

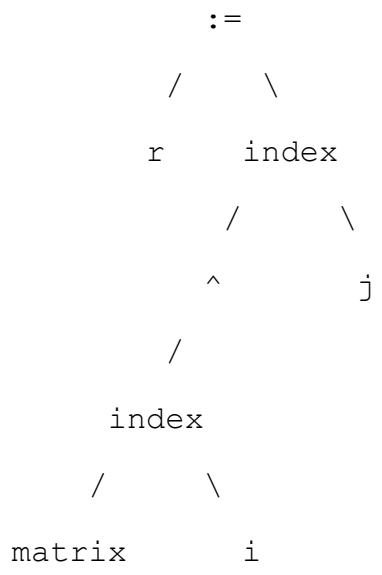
Основные проблемы модуля генерации операторов связаны с тем, что в процессе генерации некоторого оператора может возникнуть необходимость во временной переменной. Поэтому генерация оператора выполняется в два обхода (под-)дерева оператора. На первом обходе выполняется анализ оператора, описываются временные переменные и выполняется преобразование дерева, если это необходимо. На втором обходе происходит собственно генерация текста.

Рассмотрим генерацию на примере оператора присваивания:

```
r:=matrix[i]^[j];
```

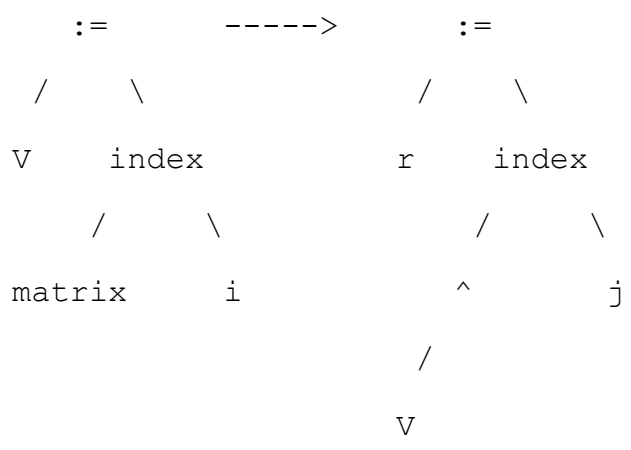
где переменные matrix, i, j, r описаны выше.

Этому оператору соответствует синтаксическое дерево:



Динамические массивы (POINTER TO ARRAY OF) представляются в виде указателя на дескриптор, содержащий длины всех измерений. Тело массива расположено непосредственно за дескриптором. Для выполнения второй индексации нам необходимо использовать указатель на дескриптор для доступа к телу массива и для проверки коректности индекса. Следовательно, необходимо сохранить этот указатель во временной переменной.

Первый обход оператора объявит временную переменную и преобразует оператор в последовательность операторов следующим образом:



где V - временная переменная типа Vector.

В результате мы получим следующий С текст:

```
{
  Vector V = matrix[X2C_CHKINX(i,8)];
  r = ((REAL *) (V+1)) [X2C_CHKINX(j,V1->Len0)];
}
```

где функция CHKINX выполняет проверку индекса массива.

Головной модуль генерации выполняет два обхода СД. Первый обход необходим для выноса вложенных процедур, так как язык С не позволяет описывать такие процедуры. На этом обходе выявляются все переменные и параметры, используемые во вложенных процедурах. Все такие переменные оформляются в отдельную структуру. Указатель на эту структуру передается дополнительно всем выносимым процедурам. Так как это решение непригодно для параметров, то все используемые параметры также передаются дополнительно. Проиллюстрируем действие алгоритма выноса на следующем примере:

```
PROCEDURE proc(a,b,c: INTEGER);

  VAR x,y,z: INTEGER;

  PROCEDURE local_1;
  BEGIN
    b:=z; -- используется переменная z и параметр b
  END local_1;

  PROCEDURE local_2;
  BEGIN
    local_1;
    x:=c; -- используется переменная x и параметр c
  END local_2;

BEGIN
  x:=y;
  local_1;
  local_2;
END proc;
```

Описание процедуры будет отображено в последовательность следующих описаний:

1) Структуры, содержащей все используемые переменные:

```
typedef struct {
  INTEGER x;
  INTEGER z;
} proc_1_NLD;
```

2) Описания локальных процедур с дополнительными параметрами – указателем на структуру и указателями на дополнительные параметры:

```
static void local_1(proc_1_NLD *NLD1,INTEGER *c,INTEGER *b)
{
  *b = NLD1->z;
}

static void local_2(proc_1_NLD *NLD1,INTEGER *b,INTEGER *c)
{
  local_1(NLD1,c,b);
  NLD1->x = (*c);
}
```

3) Описание процедуры, в которой вместо всех используемых во вложенных процедурах переменных описана переменная типа структура:

```
static void IN a_proc(INTEGER a,INTEGER b,INTEGER c)
{
  INTEGER y;
  proc_1_NLD NLD1;
  NLD1.x = y;
  local_1(&NLD1,&c,&b);
  local_2(&NLD1,&b,&c);
}
```

Такой алгоритм линеаризации процедур позволяет обеспечить независимость С текста от целевой архитектуры без потери эффективности.

Мы вкратце рассмотрели некоторые (далеко не все) проблемы генерации адекватного и эффективного текста на языке C. Задача реализации переносимого алгоритма сборки мусора для языка Оберон-2 не была нами затронута, так как эта проблема тесно связана с проблемой реализации окружения и должна решаться в рамках создания такого окружения.

По иронии судьбы, транслятор в ANSI C разрабатывался на машине, на которой нет ANSI C компилятора. Проверка полученных текстов на инструментальной машине была невозможна, и первым большим тестом для транслятора стал перенос самого себя в среду MS-DOS.

Все модули транслятора можно разделить на две существенно разные по размеру части: меньшая часть - интерфейсный и утилитный слои - является потенциально не переносимой, большая часть, а именно парсер и генератор, являются переносимы. При переносе транслятора интерфейсный и утилитный слои были переписаны на языке C. Причем переписывались только реализующие модули, а определяющие модули просто транслировались. Также на языке C была реализована библиотека вспомогательных функций и динамической поддержки. Чтобы избежать зависимости от различных реализаций языка C при реализации этих слоев, использовался минимальный набор функций из стандартизованных ANSI библиотек. Впрочем, полной независимости интерфейсного слоя достичь не удалось. Так, для реализации поиска файла по путям поиска необходимо знать разделитель в полном имени файла. Таким разделителем является символ "/" для ОС Unix, "\" для MS-DOS, а для VMS разделитель отсутствует. Для того чтобы учитывать это и другие различия, мы использовали стандартную технику: в заголовке стандартной библиотеки определяется вид платформы, и

только это определение платформы необходимо поменять при переносе транслятора.

Изучение различий и разработка переносимого интерфейсного слоя заняли довольно много времени, но в результате удалось получить продукт с очень высокими показателями переносимости. Так, в настоящее время трансляторы работают на следующих платформах: ОС Excelsior, MS-DOS, VAX/VMS (перенос выполнен автором), Sun SparcStation, HP-700, Silicon Graphics INDIGO (перенос выполнен Д.Кузнецовым). Время переноса на очередную платформу измеряется несколькими часами и, как правило, определяется скоростью чтения флоппи-диска.

2.6.3. Компиляторы для машин на базе i80386/486

Описание компиляторов можно найти в [24]. Реализация генерации кода была выполнена А.Денисовым и О.Шатохиным. В переносе компилятора активное участие принимал А.Хапугин. Он же реализовал модуль динамической поддержки для языка Оберон-2.

Одним из наиболее существенных вопросов при разработке нового компилятора является выбор целевой исполняющей системы, то есть выбор модели задачи, в рамках которой исполняется порожденный компилятором код. Традиционным решением для процессоров серии i80x86 является генерация кода фактически для самого младшего процессора семейства ("виртуальный 8086"), создание стандартного загрузочного или объектного модуля (.EXE- или .OBJ- файла) и использование штатных механизмов ОС MS-DOS по распределению ресурсов.

Однако для целей нашего проекта путь этот представляется неудовлетворительным, так как он не позволяет полностью

использовать возможности, предоставляемые старшими моделями семейства - i80386/i80486.

Было принято решение изначально ориентироваться на модель задачи, способной исполняться только на процессорах i80386/i80486 и достаточно полно использовать такие предоставляемые аппаратурой возможности, как 32-разрядная арифметика, наличие 4-Гбайтного адресного пространства, большая физическая память, страничный диспетчер, механизм защиты. В силу этого задачи должны исполняться в защищенном режиме процессора. При этом используется "почти плоская" модель памяти, то есть используются только два сегмента памяти, один из которых содержит данные всех модулей, а другой - коды всех модулей задачи.

При генерации кода используется размещение всех объектов модуля только в памяти, передача процедурных параметров производится целиком через процедурный стек задачи. Все внешние объекты адресуются косвенно через таблицу внешних ссылок, хранящуюся для каждого модуля в его глобальной области. Такое решение позволяет обходиться при связывании/загрузке модуля настройкой лишь нескольких его таблиц (внешних ссылок, CASE-переходов). За счет использования механизма временных переменных и динамического размещения собственных рабочих структур кодогенератор не имеет внутренних ограничений на сложность компилируемой программы.

Кроме компилятора был реализован специальный загрузчик, который переводит процессор в защищенный режим (protected mode PM), после чего взаимодействие с ОС происходит путем "ретрансляции" аппаратных и системных прерываний в незащищенный режим (real mode RM), т.е. загрузчик переключает процессор в RM,

давая возможность ОС выполнить свои функции в RM, затем возбуждает прерывание с таким же номером, после чего возвращает процессор в PM для продолжения исполнения загруженной задачи. Время, затрачиваемое на переключение процессора в RM и возврат его в PM, пренебрежимо мало по сравнению со временем обработки большинства аппаратных и системных прерываний, поэтому производительность системы практически не снижается.

Для получения информации о наличии и использовании расширенной памяти загрузчик обращается к драйверу расширенной памяти (extended memory manager XMM), что исключает возможные конфликты с иным програмным обеспечением MS-DOS, использующим расширенную память. Вся полученная таким образом память с помощью страничного диспетчера "выстраивается" в линейном пространстве адресов в непрерывный участок памяти, начинающийся с некоторого фиксированного адреса. Это позволяет избежать дополнительной настройки образа задачи после его загрузки и устраняет влияние фрагментации памяти в момент запуска задачи.

2.7. Особенности реализации языка Оберон-2

Необходимо отметить две существенные особенности реализации языка Оберон-2:

- канонизация симфайла;
- динамическая типизация и вызов методов.

2.7.1. Канонизация симфайла

В языке Оберон-2 нет разделения единицы компиляции на определяющую и реализующую часть. Вместо этого в тексте модуля все экспортируемые объекты должны быть помечены символами "*"

или "-" (экспорт только для чтения). Для того, чтобы избежать перекомпиляции клиентов (модулей, импортирующих данный), после каждой компиляции модуля компилятор сравнивает новый симфайл, полученный при данной компиляции, со старым симфайлом. Если новый и старый симфайлы различаются, то старый симфайл заменяется на новый. Если же они совпадают (то есть список экспорта не изменился), то удаляется новый симфайл.

Вообще говоря, для сравнения симфайлов требуется сравнить два (возможно циклических) графа. Проблема эта не является неразрешимой, но во всех известных нам компиляторах используется другой, более простой способ (см., например, [23]). При построении симфайла используется некоторый алгоритм канонизации (получения канонического симфайла). После этого два файла сравниваются байт за байтом. Если файлы совпадают, то совпадают и все экспортируемые объекты.

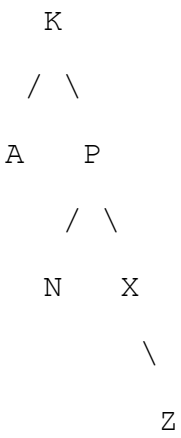
Алгоритм канонизации должен обеспечить устойчивость симфайла к несущественным изменениям текста, например, к изменению порядка описания процедур или типов, или перестановке неэкспортируемых переменных или полей записи и т.д.

Традиционно используется простой алгоритм записи объектов в алфавитном порядке по их именам. Такой алгоритм легко реализуется, если таблица символов построена на упорядоченных бинарных деревьях. К сожалению, при чтении такого симфайла получается вырожденное бинарное дерево (линейный список). Заметим, что внешних объектов может быть достаточно много, особенно по сравнению с локальными объектами, и следовательно, уровень сбалансированности дерева внешних объектов существенно влияет на скорость компиляции.

Для того, чтобы избежать вырождения дерева в компиляторе используется оригинальный алгоритм канонизации, который записывает объекты в порядке инфиксного обхода сбалансированного дерева. При чтении симфайла не делается никаких дополнительных действий. Дерево видимости, построенное по симфайлу, является сбалансированным.

Алгоритм канонизации выполняется в два этапа. На первом строится линейный упорядоченный список всех объектов и вычисляется число объектов. На втором этапе этот список преобразуется в список, упорядоченный нужным нам образом. Время выполнения каждого этапа зависит линейно от числа объектов.

Например, пусть дерево видимости выглядит следующим образом:



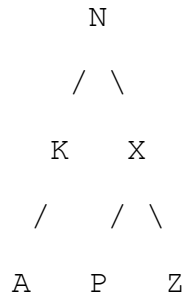
На первом этапе дерево будет перестроено в линейный список, с помощью инфиксного рекурсивного обхода.

A -> K -> N -> P -> X -> Z

Далее список будет перестроен, и объекты будут записаны в симфайл в следующем порядке:

N -> K -> A -> X -> P -> Z

И при чтении симфайла мы получим сбалансированное дерево:



2.7.2. Динамическая типизация и вызов методов

В работе [23] описана структура дескриптора типа, позволяющая выполнять динамическую типизацию и вызов методов за константное время. Дескриптор типа состоит из таблицы указателей на дескрипторы базовых типов (фиксированного размера), таблицы методов и некоторой дополнительной информации, необходимой для работы алгоритма сборки мусора. Для каждого типа статически известен уровень типизации, и динамическая проверка типа p IS T транслируется в следующее действие:

```
дескриптор(p)^.таблица_типов[уровень_типа(T)] = дескриптор(T)
```

То есть, динамический тип p является расширением типа T, если его предок уровня типа T равен T.

Дескрипторы всех типов записи модуля строятся при инициализации данного модуля процедурой динамической поддержки. Параметром этой процедуры является образ типовой системы модуля, построенный компилятором. Образ содержит описатели всех типов записи модуля, описатели команд (экспортируемых процедур без параметров) и описатель модуля.

Описатель типа записи содержит следующую информацию:

- размер записи;
- ссылку на базовый тип записи;

- число методов;
- описатели методов;
- смещения всех полей указательного и процедурного типов.

Размер записи используется процедурами выделения памяти и сборки мусора. Ссылка на базовый тип записи позволяет сформировать таблицу типов. Описатели методов определяют таблицу методов. Информация о размещении полей указательного типа используется в фазе маркировки алгоритма сборки мусора.

Для глобальных переменных модуля строится описатель модуля, содержащий точно такую же информацию, как и описатель записи, так как глобальную область естественно рассматривать как аналог записи.

2.8. Характеристики семейства и сравнение его представителей

Еще раз отметим основные особенности реализации семейства:

- тесно связанная реализация двух языков позволяет разрабатывать ПО на двух языках, используя в каждом случае достоинства каждого языка;
- промежуточное представление является объединением представления двух языков;
- только синтаксический анализ реализован отдельно для каждого из входных языков, все остальные компоненты компиляторов являются общими;
- синтаксический анализ выполнен методом рекурсивного спуска;
- для реализации таблицы символов и алгоритмов идентификации используются бинарные деревья;

- при записи симфайла используется оригинальный алгоритм канонизации дерева видимости;
- модель взаимодействия парсера и генератора позволяет получить полностью независимый от платформы парсер;
- наличие в составе семейства транслятора в ANSI C позволяет осуществить быстрый перенос ПО практически на любую платформу;
- все семейство реализовано на языке Модула-2.

Приведем далее некоторые характеристики компиляторов.

Размеры модулей парсера указаны в строках. В скобках приведены размеры определяющего модуля.

определение внутреннего представления	45(270)
лексический анализ	480(90)
видимость, чтение/запись симфайлов	1270(120)
построение дерева, семантический контроль	1800(100)
синтаксический анализ (Модула-2)	1750(15)
синтаксический анализ (Оберон-2)	1535(15)

Итого, размер фазы анализа для языка Модула-2 - 5345(595) строк, а для языка Оберон-2 - 5130(595). Причем, общая часть составляет 3595(580) строк. Небольшие размеры фазы анализа подтверждают правильность выбора схемы трансляции.

Далее проведем сравнение различных генераций:

Характеристика	Кронос	ANSI C	i386/486
Размер генератора (в строках)	4600	4100	11000
Время разработки (человекомесяцев)	3	8	24
Скорость компиляции (строк/минуту)	2000	8000	30000
оценка скорости ЭВМ (Dhrystones/sec)	1200 (Kronos-2.6WS)	4200 (i386 14Mhz)	25000 (i496 33Mhz)
Время самокомпиляции (секунды)	600	70	40

Успешный перенос компиляторов и их использование дает нам основание утверждать, что идеи, заложенные в основу семейства, являются весьма продуктивными. Дальнейшее развитие семейства может идти в двух направлениях: перенос на другие платформы и реализация машинно-независимых (или машинно-ориентированных) оптимизирующих преобразователей дерева. Мы надеемся, что движение будет происходить в обоих направлениях.

3. Расширяемые системы на примере системы Оберон

В этой главе мы рассмотрим принципы построения РПС на примере системы Оберон [25]. Кроме собственно системы Оберон мы будем рассматривать расширяемый редактор Write [26], реализованный как одно из приложений системы.

Описывая систему Оберон мы постараемся определить базовые понятия и дать читателю основу для сравнения при описании системы Мифрил (гл. 4).

Разработка системы Оберон была начата Н.Виртом и Ю.Гуткнехтом в 1985 году. Целью проекта была разработка современной и переносимой ОС для персональных рабочих станций. Одновременно с разработкой системы был спроектирован и реализован язык Оберон. В первую очередь система и Оберон компилятор были реализованы для станций Ceres [27]. Далее был выполнен перенос на такие платформы, как Mac II [15], SparcStation/Unix [14], DecStation/Unix, i386/AIX, Chameleon [28]. На всех этих платформах (кроме Ceres и Chameleon) система Оберон не является операционной системой в традиционном смысле, а некоторым окружением, работающем над базовой ОС. Сравнение различных реализаций системы приводится в [29].

3.1. Обзор системы

Система Оберон является однопользовательской, однопроцессной и многозадачной системой. Говоря о многозадачности, мы, вслед за Н.Виртом, имеем в виду то, что пользователь может свободно переключаться с одной деятельности на другую, не запуская новых утилит.

Для работы система необходим графический экран и мышь. Экран разделен на вертикальные полосы (Tracks). Стандартная конфигурация экрана состоит из двух полос: пользовательской и системной. Каждая полоса, в свою очередь, состоит из нескольких окон (Viewer). Система не поддерживает концепцию перекрывающихся окон. Ширина всех окон равна ширине полосы, а при изменении вертикального размера некоторого окна происходит соответствующие изменения прилежащих окон. Полоса может быть целиком накрыта другой полосой. Такое строение окон является достаточно удобным, хотя и непривычным. Заметим только, что для экранов небольших размеров такая структура окон является мало пригодной.

При запуске системы пользователь видит на экране окно стандартного вывода и окно (System.Tool), содержащее набор основных команд (в том числе команду создания нового окна). Каждая команда M.P состоит из имени модуля M и имени процедуры без параметров P. Пользователь запускает команду, нажимая среднюю кнопку мыши на тексте команды. При активации команды система вызывает процедуру P из модуля M (загружая модуль M, если этот модуль не был загружен в систему).

Исполнение команды – основной способом взаимодействия пользователя с системой. Набор команд включает в себя такие команды, как "открытие окна", "запись окна", "распечатка директории" и многие другие.

Каждое изменение состояния мыши приводит к вызову процедуры управления данным окном (тем, на которое указывает курсор). Окна разного вида могут различным образом реагировать на нажатие кнопки. Особый интерес представляют собой текстовые окна, то есть окна, изображающие некоторый текст. Над текстом определен стандартный набор операций редактирования. Для каждого символа в

тексте определены атрибуты: цвет (интенсивность для ч/б мониторов), фонт и вертикальное смещение.

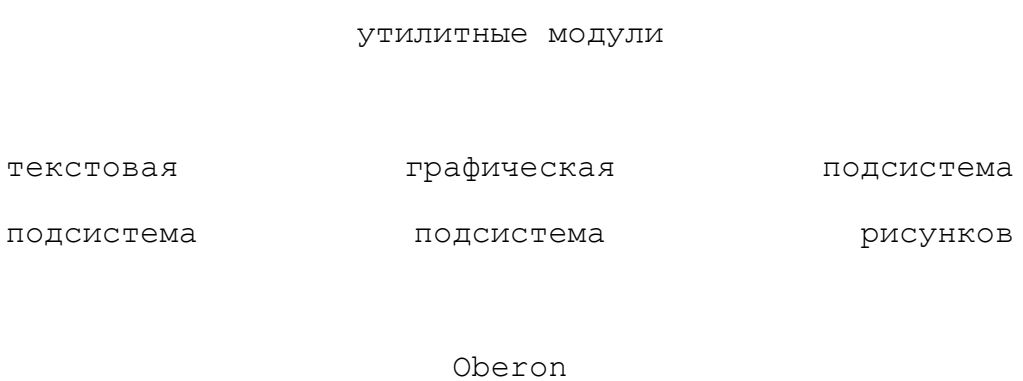
В любом текстовом окно пользователь может написать текст команды (M.P) и после этого запустить ее. Стандартный набор команд для некоторой деятельности может быть записан в файл. После чего достаточно открыть окно, изображающее этот файл, для того чтобы исполнять команды. Система содержит набор таких файлов, содержащих команды (tool packages):

Edit.Tool	-	создание и редактирование текстовых окон
Draw.Tool	-	создание и редактирование графических окон
System.Tool	-	определение и изменение конфигурации, файловые операции, ...
Compiler.Tool	-	Оберон компилятор
Net.Tool	-	работа с сетью и электронной почтой

Заметим, что все эти файлы - это просто текстовые файлы с именами команд.

3.2. Структура системы

Система Оберон состоит из следующих компонент: внутреннее ядро, внешнее ядро, подсистемы текстов, графики, рисунков и набора утилитных модулей (tools). Утилитный модуль - это модуль, реализующий набор команд.



Внутреннее ядро реализует операции работы с памятью, файлами и динамический загрузчик модулей. Внешнее ядро реализует управление окнами (viewers), понятия текста и фонта, а также содержит набор драйверов (клавиатура, мышь, сеть, экран). Модуль Oberon определяет интерфейс между ядром системы и клиентами.

Каждая подсистема состоит из трех модулей. Рассмотрим структуру подсистемы на примере подсистемы текстов. Она состоит из следующих модулей:

Texts - определяет понятие текста

TextFrames - определяет понятие текстового окна

Edit - утилитный модуль

Модуль Texts определяет тип и набор базовых операций над ним. Модуль TextFrames реализует текстовое окно и операции редактирования. Модуль Edit содержит набор команд открытия/записи и дополнительные операции редактирования.

Как и в любой другой системе, в системе Оберон имеется набор приложений, увеличивающих функциональные возможности.

Важно заметить, что в системе нет разницы между собственно системой и приложениями. Каждое приложение определяет свой набор команд, и различные приложения могут использоваться одновременно. Мы упомянем только некоторые из реализованных приложений: редактор гипертекста [30], расширяемый редактор [26], графический редактор и система трассировки [31], которые использовались при разработке рабочей станции Ceres-3.

3.3. Возможности расширения

Функциональные возможности системы могут увеличиваться двумя способами: реализация новых команд и расширение базовых понятий.

Реализация новых команд

Для реализации новой команды достаточно написать и скомпилировать модуль, содержащий экспортируемую процедуру без параметров. Например:

```
MODULE World;

IMPORT Texts, Oberon;

VAR
  i: LONGINT;
  W: Texts.Writer;

PROCEDURE Hello*;
```

```
BEGIN
  Texts.WriteInt(W,i,0);
  Texts.WriteString(W,": Hello, world");
  Texts.WriteLn(W);
  Texts.Append(Oberon.Log,W.buf);
  INC(i);
END Hello;

BEGIN
  i:=1;
  Texts.OpenWriter(W); -- открыли буфер записи
  Texts.WriteString(W,"What a wonderful world!");
  Texts.WriteLn(W);
  Texts.Append(Oberon.Log,W.buf); -- добавили буфер к окну
                                  -- стандартного вывода
END World.
```

После компиляции модуля напишем в любом текстовом окне "World.Hello" и запустим команду. Модуль "World" загрузится и напечатает "What a wonderful world!" и "1: Hello, world" в окно стандартного вывода. При следующих запусках команды будем печататься только строка "Hello, world", префиксированная номером запуска. Имя команды может быть написано в любом месте, например последняя строка нашего модуля может выглядеть так:

```
END World.Hello (* It's fun, isn't it? *)
```

Если в модуле сделаны некоторые изменения, и он заново скомпилирован, можно запустить новую версию модуля, нажав

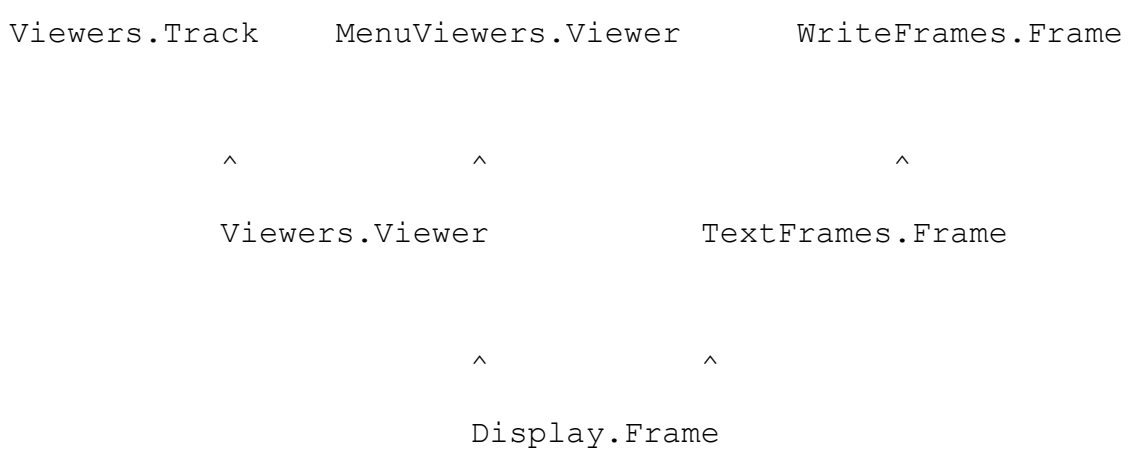
комбинацию кнопок мыши, или предварительно исполнив команду выгрузки (System.Free). Каждый раз при загрузке модуля будет выполнена его инициализация.

Отметим несколько важных особенностей такого подхода:

- новая команда ничем не отличается от любых других;
- при запуске команды загружаются все модули, ее реализуюшие;
- загруженный модуль остается в памяти, до его явной выгрузки, и все глобальные данные сохраняются между вызовами команд, таким образом, при работе системе создается среда, содержащая только используемые пользователем команды;
- для переключений с одного вида деятельности к другому не нужно выходить из одной системы и входить в другую;
- определен набор команд, позволяющих узнать имена загруженных модулей и имена всех команд данного модуля.

Расширение базовых понятий

Рассмотрим механизм расширения понятия на примере типа "фрейм" (Frame) системы Оберон. Определение нового (расширенного) типа фрейм является стандартным методом расширения системы. Система включает большой набор таких расширений. Следующая картинка показывает (неполное) дерево основных расширений (узел дерева имеет вид ИмяМодуля.ИмяТипа).



Другими расширениями типа Display.Frame являются

GraphicFrames.Frame и

PictureFrames.Frame.

Кроме того, можно определить другие расширения, как базового типа, так и любого расширенного.

Тип Display.Frame определяет объект, описывающий координаты и размеры прямоугольника и содержащий процедуру управления.

```
TYPE
  Frame = POINTER TO FrameDesc;
  FrameDesc = RECORD
    dsc,next: Frame; (* son, brother *)
    X,Y,W,H: INTEGER;
    handle: Handler;
  END;

  Handler = PROCEDURE (Frame, VAR FrameMsg);
  FrameMsg = RECORD END;
```

Так как язык Оберон не поддерживает понятие метода, то объект содержит процедурную переменную. При создании объекта это

поле инициализируется подходящей процедурой. Таким образом, мы получаем достаточно нестандартный подход к ООП, так как различные экземпляры одного типа могут реагировать на разные наборы сообщений.

Для определения сообщения нам достаточно расширить тип FrameMsg. Например, сообщение о вводе с клавиатуры или мыши определяется в модуле Oberon следующим образом:

```
  InputMsg = RECORD (Display.FrameMsg)
    id: INTEGER; (* operation *)
    modes,keys: SET; (* mouse *)
    X,Y: INTEGER; (* position *)
    ch: CHAR; (* character read *)
  END;
```

Здесь id может принимать значения 0 (мышь) и 1 (клавиатура).

Теперь мы можем реализовать реакцию на ввод для некоторого конкретного типа (класса) фрейма.

```
PROCEDURE Handle(F: Display.Frame; VAR M: Display.FrameMsg);
BEGIN
  IF M IS Oberon.InputMsg THEN
  WITH M: Oberon.InputMsg DO
    IF M.id = 0 THEN (* mouse *)
    ...
    ELSIF M.id = 1 THEN (* keyboard *)
    ...
    END;
  ELSIF M IS ...
  END;
END Handle;
```

Типы сообщений (т.е. расширения типа FrameMsg) могут быть определены в любом модуле и, как правило, определяются в месте первого использования. Кроме сообщения о вводе, система определяет набор сообщений об изменении размера и местоположения фреймов, создании копии фреймов и окон и т.д.

Очень важным свойством использования таких сообщений является возможность определить операции оповещения. Например, процедура Broadcast позволяет послать некоторое сообщение всем фреймам:

```
PROCEDURE Broadcast(VAR M: Display.FrameMsg);
BEGIN
  FOR v IN all_visible_frames DO
    v.handle(v,M)
  END;
END Broadcast.
```

Вернемся к иерархии окон. Модуль Viewers определяет тип Viewer - базовое понятие окна, и тип Track - окно, заполняющее на экране вертикальную полосу. Модуль MenuViewers определяет окно стандартного вида, состоящего из двух фреймов:

Меню-фрейм является текстовым окном, а рабочее окно – любым расширением фрейма (например, текстовым, графическим и т.д.). Модуль TextFrames определяет понятие текстового фрейма, а модуль WriteFrames расширяет это понятие.

Для расширения фрейма необходимо реализовать управляющую процедуру, реагирующую на сообщения системы. Как правило, каждое расширение фрейма изображает и позволяет редактировать некоторую модель (текст для текстового фрейма и т.д.).

Рассмотрим более подробно использование механизма расширения типов для наращивания мощности редактора Write [26]. Редактор состоит из модулей, реализующих расширение стандартного текста, расширение текстового фрейма и утилитного модуля, позволяющего открыть расширенный текстовый фрейм. Редактор реализует некоторый стандартный набор операций, но в отличие от обычных редакторов, текст может содержать абстрактную литеру, так называемый элемент.

Каждый элемент - это конкретизация абстрактного класса, содержащего набор методов, необходимых для изображения, копирования, реакции на ввод, сохранения в файле и т.д. Редактор, выполняя некоторую операцию над фрагментом текста, содержащим элемент (например, копирование) посылает элементу нужное сообщение.

Расширяя элемент, мы можем получить текст, содержащий картинки, графики, диаграммы, таблицы и т.д. Особую роль в реализации редактора играет динамическая загрузка модулей. В каждый момент в системе подгружены только модули, реализующие используемые элементы. Чтобы дать читателю некоторое представление о мощности данного подхода, перечислим некоторые из реализованных расширений:

1) Элементы, определяющие структуру текста

Единственный элемент, встроенный в редактор. Позволяет определить ширину строки, способ выравнивания и другие атрибуты форматирования. Изображается в виде горизонтальной линии с дополнительными маркерами. С помощью мыши можно изменить атрибуты параграфа. Установленные элементом атрибуты форматирования действуют на текст до следующего такого элемента.

2) Элементы – маркеры ошибок

Вставляются компилятором в те позиции текста, где обнаружена ошибка. В исходном виде элемент изображается в виде квадратика. При нажатии кнопки мыши элемент раскрывается в прямоугольник, содержащий сообщение об ошибке. Дополнительный набор команд позволяет перейти к следующему элементу и удалить все такие элементы из текста. Элемент является временным, то есть не записывается в файл при записи текста.

3) Активные элементы (Clock Elements, Icon Elements)

Эти элементы являются активными, так как реагируют на внешнюю по отношению к редактору активность, а именно на системный счетчик времени. При этом часы идут и показывают точное время, а на иконке изображен шагающий человечек. С помощью операции редактирования мы можем заполнить все окно шагающими человечками, и в то же время это будет нормальный, редактируемый текст, который можно записать в файл или напечатать. К

сожалению, в настоящее время не существует технологии, позволяющей изображать движущиеся фигурки на листе бумаги.

4) Элементы - меню (PopUp Elements)

Изображается прямоугольником с некоторым текстом. При нажатии кнопки мыши прямоугольник раскрывается в меню, содержащее набор команд.

5) Графические элементы и элементы-рисунки

Изображают некоторый графический объект или рисунок. При нажатии кнопки мыши открывается окно стандартного редактора.

6) Структурные элементы (Fold Elements)

Позволяют скрывать фрагменты текста и, тем самым, поддерживают различные способы изображения текста. Например, можно спрятать реализацию всех процедур, получив аналог определяющего модуля.

Для реализации нового элемента необходимо расширить тип, реализовать процедуру управления элементом, команду вставки элемента и набор команд, специфичных для данного элемента.

Ядро редактора Write состоит из менее чем 2000 строк на языке Оберон, и определяет понятие Текста, расширяя стандартный текст добавлением элементов, и понятие текстового фрейма, расширяя стандартный добавлением операций над элементами. Реализация элементов позволила превратить Write в очень мощный и универсальный редактор. Он позволяет редактировать как тексты

программ, так и готовить документацию. Простая модель элемента позволила сделать достаточно много, хотя некоторые возможности системы подготовки документации достаточно трудно реализовать в рамках этой простой модели (например, обтекание текста вокруг рисунка или фотографии).

3.4. Анализ системы

Расширяемая система содержит описания одного или несколько типов, являющихся базой для дальнейших расширений. Очень важным является выбор правильного набора абстрактных операций. В некоторых случаях такой набор операций является достаточно очевидным (например, в случае элемента), в других - очень сложным.

Система Оберон фактически содержит один тип, являющийся базой для расширения (Frame). Остальные типы (Текст, Файл, Фонт) тоже могут быть расширены, но все операции над этими типами реализованы в виде статических процедур (не методов). Поэтому все операции над расширенным объектом должны импортироваться из другого модуля, и могут возникнуть некоторые проблемы при применении базовой операции к расширенному типу (что допускается правилами совместимости типов). Так например, если записать текст, содержащий элементы (WriteTextsxt) процедурой из модуля Texts, то все элементы будут утеряны. В документации по редактору Write перечислены все базовые операции, которые нельзя применять к расширенному типу. Повторим еще раз, что типовой контроль в данном случае невозможен. Реализация базовых понятий с помощью методов (Oberon-2 type-bound procedures) позволяет

решить эту проблему, так как операция всегда соответствует типу (уровню расширения типа).

Отсутствие методов также приводит к тому, что у объекта есть только одна управляющая процедура (Handle), и тело этой процедуры состоит из выбора по типу объекта и по типу сообщения. Использование методов позволило бы определить отдельный метод для каждой операции и повысить эффективность.

Кроме недостатков системы, связанных с возможностями расширения, хотелось бы отметить еще две особенности системы: низкий уровень графики и отсутствие перекрывающихся окон.

4. Система Мифрил

В данной главе мы рассмотрим принципы построения, структуру и возможности расширения системы Мифрил (см. также [32]). Название системы взято из [33]. Мифрил - это название подлинного серебра (the True Silver), которое добывалось гномами в недрах Мории. Удивительно легкий и прочный метал обогатил и прославил гномов.

Мы полагаем, что слово Mithril сконструировано Толкиеном из словосочетания mythical rill (мифический источник), и это расшифровка очень подходит в качестве имени РПС.

Мы предполагаем, что система Mithril будет использоваться как база для разработки переносимого прикладного ПО. Для того чтобы упростить разработку прикладного ПО необходимо решить несколько важных задач:

1) Необходимо создать комфортную обстановку для программиста (языки, отладка, поддержка проекта);

2) Система должна включать набор средств для создания пользовательского интерфейса (окна, мышь, конструктор интерфейсов);

3) Система должна поддерживать одинаковую обстановку на разных платформах (т.е. на разных машинах и под разными ОС).

С точки зрения пользователя, система практически не отличается от системы Oberon. При запуске системы пользователь видит на экране окно стандартного вывода и окно (System.Tool), содержащее набор основных команд (в том числе команду создания нового окна). Каждая команда M.P состоит из имени модуля M и имени процедуры без параметров P. Пользователь запускает команду, нажимая среднюю кнопку мыши на тексте команды. При

активации команды система вызывает процедуру P из модуля M (загружая модуль M, если этот модуль не был загружен в систему).

В отличие от системы Оберон система Mithril поддерживает универсальный многооконный интерфейс. Принятое в системе Оберон деление на "треки" может быть поддержано реализацией подходящей эвристики размещения окон на экране. Такое решение не только увеличивает свободу размещения окон на экране, но и позволяет работать в системе на экранах небольшого размера.

В каждый момент на экране расположено множество окон. Каждое окно содержит органы управления (для перемещения или изменения размеров) и несколько подокон. Основное подокно называется рабочим. Так как понятие текста и текстовых окон является очень важным (все есть текст), то основные операции текстового редактирования выполняются с помощью кнопок мыши.

4.1. Принципы построения и структура системы

Система состоит из трех частей: динамической поддержки (run time support, RTS), ядра системы и оболочки. Кроме этого система включает в себя набор драйверов для взаимодействия с операционной системой и/или аппаратурой.

Динамическая поддержка (RTS) - это неотъемлемая часть реализации языка Оберон-2, содержащая операции выделения памяти, сборки мусора и финализации объектов. В нашей системе RTS написан на языке Модула-2. При переносе системы RTS (как правило) переписывается для повышения эффективности сборки мусора.

Ядро системы определяет набор базовых понятий (типов): постоянный объект, файл, окно, текст, и т.д. (см. 4.1.2). При

запуске ядро выполняет действия, необходимые для конфигурации системы, в том числе действия по установке драйверов.

Стандартная оболочка системы реализует минимальный набор понятий, определяющих интерфейс пользователя (в первую очередь текстовые окна) и набор команд. Расширение оболочки может выполняться двумя способами: расширение набора команд (динамическое расширение) и расширение некоторого базового типа (например, расширение окна до графического окна).

4.1.1. Влияние сборки мусора на построение системы

Система, построенная с использованием сборки мусора, имеет свои особенности. Основное отличие от обычных систем состоит в том, что возвращение всех ресурсов системы также должно выполняться во время сборки мусора. Рассмотрим, например, некоторую структуру данных, содержащую дескрипторы открытых файлов. Для того чтобы определить, нужен ли еще этот файл, необходимо знать число ссылок на него, а это информация известна только при сборке мусора. Закрытие файла (т.е. уничтожение дескриптора открытого файла) может выполняться, если только на него нет ссылок. Точно так же обстоит дело и с другими ресурсами.

Следовательно, возникает необходимость в механизме финализации, то есть в возможности выполнить некоторое действие над объектом в том момент, когда на него уже нет ссылок.

В некоторых случаях мы можем спроектировать дескриптор ресурса таким образом, что возвращение этого ресурса сводится к возвращению памяти, занимаемой дескриптором, и, следовательно, финализация для такого ресурса не нужна. Таким образом

реализованы файлы в системе Оберон/Ceres. Но при переносе системы под Unix (см. [14]) возникла необходимость финализации файлов, так как число одновременно открытых файлов в системе Unix ограничено.

Для того чтобы механизм финализации работал, система не должна содержать дополнительных указателей на дескрипторы ресурсов, иначе число ссылок никогда не станет равным нулю. Это требование вступает в противоречие с требованием обхода всех дескрипторов ресурса, что часто нужно для того, чтобы одному ресурсу соответствовало не более одного дескриптора.

Выход из этого положения прост: достаточно связать все дескрипторы ресурсов ссылками, которые не являются ссылками с точки зрения сборщика мусора. В языке Модула-3 эта возможность реализуется специальным ключевым словом UNTRACED:

```
TYPE Untraced = UNTRACED POINTER TO X;
```

Мы, вслед за [13], реализуем эту возможность с помощью специальных указаний компилятору (системных флагов). При описании типа можно указать системный флаг (целое число), который определяет специфику данного типа. Эта возможность используется не только для получения нетрассируемых указателей, но и в тех случаях, когда для интерфейса с ОС или аппаратурой необходимо изменить способ размещения типа (см. также [15]).

Пример:

```
TYPE Untraced = POINTER TO [1] X;
```

Заметим, что есть другой способ получить такой указатель в нашей биязыковой системе - описать его в определяющем модуле на

языке Модула-2 и проимпортировать в модуль, написанный на языке Оберон-2.

Для того чтобы в разных частях системы не нужно было пользоваться низкоуровневой возможностью мы (на нижнем уровне системы) определяем понятие линейного нетрассируемого списка с операцией вставки. Каждый такой список содержит процедуру финализации (пустую, если финализация не нужна). Глобальным объектом сборщика мусора является нетрассируемый список таких списков (мета-список). Сборщик мусора перед возвращением памяти обходит список и вызывает процедуру финализации для всех "мусорных" объектов. Идея совместной организации механизма финализации и нетрассируемых ссылок описана в [34].

Модули системы, реализующие некоторый ресурс, должны создать заголовок списка и указать процедуру финализации, а при создании нового дескриптора ресурса – вставить его в этот список.

4.1.2. Возможности расширения и иерархия базовых типов

Как и в системе Оберон, имеется два способа расширения системы: расширение набора команд и расширение базовых понятий. Кроме того, можно добавить к системе некоторое новое понятие, независимое от предыдущих, но такая возможность существует в любой системе, и мы не будем ее рассматривать. Для реализации новой команды нужно выполнить те же действия, что и в системе Оберон.

Как известно из предыдущих глав, расширяемая система должна определять набор базовых понятий (типов или классов). Выбор

базовых понятий и методов в большой степени определяет потенциал системы.

Кратко перечислим основные понятия системы Мифрил (подробнее см. 4.3). Мы будем приводить список понятий (типов) в следующей нотации

Тип (БазовыйТип [,Расширение])

Здесь Тип есть ИмяМодуля.ИмяТипа, БазовыйТип - это имя типа или NIL (если сам тип является базовым), а Расширение может принимать одно из следующих значений:

Class - тип используется только как базовый для расширения;

Leaf - тип является листом в иерархии типов, то есть расширение его или запрещено или не поддерживается.

Если Расширение не указано, то тип является конкретным, то есть его методы реализованы, и в то же время он может быть расширен. Заметим, что на языке Оберон мы можем описать тип, расширение которого запрещено, вне модуля в котором этот тип определяется.

Иерархия типов системы Мифрил

Objects.Object (NIL,Class)

Пустой объект - корень дерева типов. Практически все типы системы являются его расширениями, что позволяет строить гетерогенные структуры. Для любого расширения такого объекта может быть определена операция финализации.

Errors.Error (Objects.Object)

Errors.Exception (Objects.Object)

Определяют контекст ошибки и реакцию на ошибку.

Channels.Channel (Objects.Object,Class)

Channels.Rider (Objects.Object,Class)

Определяют понятие канала ввода/вывода (файл, терминал, сеть, ...) и объекта доступа (см. также [34]).

XRiders.Rider (Channels.Rider,Leaf)

Определяет стандартный объект доступа. Позволяет читать и писать данные в машинно-независимом формате.

Storage.Object (Objects.Object,Class)

Определяет постоянный объект, то есть объект, состояния которого может быть записано в файл и восстановлено из файла.

Files.File (Channels.Channel,Leaf)

Файл.

Fonts.Font (Storage.Object,Class)

Определяет абстрактный фонт.

Texts.Text (Storage.Object)

Texts.Elem (Storage.Object)

Текст со встроенными абстрактными литерами (элементами).

Screens.Screen (Storage.Object,Class)

Определяет стандарт драйвера экрана (см. 4.1.3).

Windows.Window (Storage.Object)

Окно.

Перечисленный набор типов является базовым. Оболочка системы в основном расширяет типы Windows.Window и Texts.Elem. Расширение типов Fonts.Font и Screens.Screen задают конкретизацию этих понятий и используются для настройки системы на структуру фонта и на вид экрана.

Важным вопросом при определении базовых типов является выбор вида операций над объектом. Язык Оберон-2 позволяет использовать три вида операций: статические процедуры, методы (процедуры, ассоциированные с типом) и процедурные переменные. Кроме того, если некоторый параметр операции есть запись, передаваемая по ссылке, то вместо этого параметра может передаваться любое ее расширение, и тем самым мы получаем способ передачи сообщений, характерный для языка и системы Оберон.

Мы использовали следующие принципы организации интерфейса объекта:

1) Методы используются при описании абстрактного класса. Для каждой операции используется отдельный метод. Дополнительный метод (handle) используется для передачи сообщений.

2) Операции оповещения реализуются также как в системе Оберон, то есть через сообщения.

3) Операции, специфичные для типов-листов, оформляются в виде статических процедур.

4) Мы старались исключить использование процедурных переменных, заменяя их парой объект-метод, так как процедурные переменные потенциально опасны при выгрузке модуля; кроме того использование объекта вместо процедурного значение позволяет передавать контекст операции (см. также итерацию директории в 4.3.6).

Выбор способа итерации скрытых структур также является очень важным. Система содержит достаточно много таких структур, например: множество элементов в тексте, множество загруженных модулей, множество файлов в директории. Мы намеренно используем слово "множество" (вместо слова "список"), чтобы указать на недоступность внутренней структуры. Скрытость структуры обеспечивает надежность, но при этом должна быть предусмотрена возможность обхода такой структуры и выполнения некоторой операции над всеми ее объектами (например, для того, чтобы напечатать список файлов).

При итерации особое внимание необходимо уделять целостности структуры. Что произойдет, если во время итерации директории будут создаваться или удаляться файлы?

Для обеспечения целостности итерация может выполняться на копии структуры или просто выдавать копию структуры (вернее ее

публичной части). В каждом конкретном случае (см. 4.3) мы постараемся дать обоснование выбранному методу.

4.1.3. Повышение переносимости и адаптируемости

Для повышения переносимости системы необходимо выдержать разделение ее на переносимые и системно-зависимые (или машинно-зависимые) части. Мы предполагаем, что система Мифрил будет работать в основном над некоторой стандартной ОС, или некоторым стандартным окружением (например, X Windows). Часть модулей ядра (Файлы, Загрузчик) зависят от системы и должны быть модифицированы при переносе. Для таких модулей важно спроектировать интерфейс, независимый от платформы. Интерфейс должен соответствовать двум (противоречивым) требованиям:

- полнота, т.е. интерфейс определяет функции для всех стандартных действий;
- реализуемость, т.е. должна существовать возможность реализовать все функции на любой платформе.

Рассмотрим подробнее, как эти требования могут быть удовлетворены на примере модуля Files. Остановимся на двух различиях в реализации файлов в системах Unix и MS-DOS.

В ОС Unix (и в ОС Excelsior) принято отделять понятие физического файла от его именования. Одному физическому файлу может соответствовать несколько имен. Операция именования файла (т.е. добавления еще одного имени) и операция удаления имени отделены от операций создания/удаления файла.

В MS-DOS каждый файл изначально именован и нет возможности выделить операции над именами.

При проектировании интерфейса мы вынуждены ориентироваться на систему с более слабыми возможностями, и, следовательно, в системе Мифрил нельзя стандартными средствами получить файл с несколькими именами. В то же время понятие временных файлов может быть выражено в MS-DOS и поэтому включено в интерфейс модуля.

Другое различие в этих системах связано с понятием защиты файла, которая отсутствует в MS-DOS. Мы полагаем, что операции модификации битов защиты файла являются достаточно специфичными и могут быть реализованы в отдельных системно-зависимых модулях. Поэтому такие операции не входят в интерфейс модуля Files.

Кроме собственно понятия переносимости необходимо рассмотреть также пути повышения адаптируемости системы к различному оборудованию или различным стандартам. Так, например, на одной платформе могут быть различные экраны (разрешение, цветной/черно-белый, ...), и система должна работать на различных экранах (возможно, после выполнения переконфигурации).

Для повышения адаптируемости мы вводим понятие драйвера. Абстрактный драйвер - это абстрактный класс, определяющий параметры устройства и набор операции над ним. Для каждого конкретного устройства определяется расширение абстрактного драйвера, выставляющее атрибуты драйвера и реализующее операции. Конкретный драйвер может определяться в процессе конфигурации системы или подключаться к системе динамически, с использованием стандартного механизма динамической загрузки.

Ядро системы определяет два абстрактных драйвера: драйвер экрана (Screens.Screen) и драйвер фонта (Fonts.Font). Конкретный драйвер экрана реализует графические примитивы (точка, линия, окружность, символ) для конкретного экрана. Каждое окно содержит

ссылку на драйвер того экрана, на котором оно расположено, и выполняет все графические операции через операции этого драйвера. Это позволяет получить полностью независимый от аппаратуры механизм управления окнами.

Точно также введение драйвера фонта, позволяет получить независимую от структуры и вида фонта систему. На различных платформах система может использовать различные виды фонтов.

Введение понятие драйвера вносит в систему еще одно важное свойство, система может одновременно использовать различные виды фонтов и работать на нескольких экранах. Последнее свойство потребовало реализации дополнительной поддержки (см. подробнее 4.3.9).

4.2. Динамическая поддержка

Модуль динамической поддержки реализует операции выделения памяти, финализации и сборки мусора. Реализация модуля для систем Mithril/Excelsior и Mithril/MS-DOS выполнена А. Хапугиным.

Для выделения памяти и для сборки мусора используется информация о типовой системе, порожденная компилятором с языка Оберон-2. Модуль динамической поддержки неявно импортируется в любой модуль на языке Оберон-2. При инициализации любого модуля вызывается процедура, которая по информации, порожденной компилятором, строит некоторое внутреннее представление, удобное для использования.

Типовая система включает в себя информацию обо всех типах записей, методах, смещениях ссылочных полей в записях, а также

информацию, необходимую для динамического вызова команд (см. также 2.7).

Модуль содержит большой набор процедур выделения памяти, включающий в себя:

- выделения памяти под запись;
- выделение памяти под статический массив;
- выделение памяти под динамический n-мерный массив;
- выделение системной памяти (т.е. памяти, которая не обслуживается сборщиком мусора).

Для каждой записи в памяти выделяется два дополнительных слова, одно из которых содержат указатель на дескриптор типа, а другое используется в фазе маркировки сборки мусора. Для каждого массива (как статического, так и динамического) также строится дескриптор специального вида.

Для сборки мусора используется mark-and-sweep алгоритм, близкий к алгоритму, описанному в [35]. Алгоритм состоит из двух фаз: фазы маркировки и фазы освобождения памяти. В фазе маркировки алгоритм обходит все доступные по указателям объекты, начиная обход с глобалов всех модулей и с локалов на процедурном стеке. Все доступные объекты помечаются. После этого выполняется финализация объектов, которые содержатся в одном из списков финализации (см. 4.1.1) и не помечены. В фазе освобождения памяти все непомеченные объекты возвращаются в список свободной памяти.

Сборка мусора может запускаться системой периодически или при нехватке некоторого ресурса (не обязательно памяти). Так, например, многие ОС (e.g. Unix) накладывают ограничения на число одновременно открытых файлов, и сборка мусора может быть запущена при открытии файла.

4.3. Ядро системы

Ядро системы представляет собой законченную программную компоненту, которая после инициализации выполняет некоторую стандартную последовательность команд. Эта последовательность состоит из команд, которые либо создают минимальный набор окон, либо восстанавливают состояние предыдущего сеанса. После этого выполняется так называемый "основной цикл". В основном цикле система ожидает одного из внешних событий (от клавиатуры, мыши, таймера, ...) и посылает сообщение некоторому окну. Основной цикл описан в 4.3.10. В остальных разделах мы подробно описываем компоненты ядра.

Все примеры мы будем приводить на некотором обероноподобном языке, так, например, описания методов будут вставлены в описания соответствующих типов. Символ "-" после имени переменной или поля означает, что данный объект экспортируется только для чтения.

4.3.1. Объекты и финализация

Как уже говорилось, тип Objects.Object является корнем иерархии типов (см. 4.1.2). Кроме этого типа, в модуле Objects определяется еще два часто используемых типа. Тип Message - это базовый тип для всех сообщений (в стиле системы Оберон), а тип String определяет динамическую строку.

Из описаний модуля Objects:

```
TYPE
  Object = POINTER TO ObjectDesc;
  ObjectDesc = RECORD END;
  Message = RECORD END;
  String = POINTER TO ARRAY OF CHAR;
```

Модуль Closure определяет понятие нетрассируемого списка (4.1.1) и операции над ним. Тип Trailer задает голову такого списка, а тип Link - его элемент. Заметим, что Link - это нетрассируемый указатель на запись.

```
TYPE
  Close = PROCEDURE (obj: Objects.Object);
  Link = POINTER TO [1] RECORD
    obj-: Objects.Object;
    next-: Link;
  END;
  Trailer = POINTER TO RECORD (Objects.ObjectDesc)
    link-: Link;
  END;

PROCEDURE new_trailer(c: Close): Trailer;
(* Создание нового списка с указанной процедурой финализации *)

PROCEDURE insert(x: Trailer; o: Objects.Object);
(* Добавление объекта к списку *)
```

Поля типов Trailer и Link экспортируются только по чтению, поэтому удаление из этого списка невозможно. Удаление из списка

выполняется при сборке мусора, если на объект нет ссылок. Заметим, что обход такого списка возможен, что позволяет, например, выполнять поиск некоторого объекта в нетрассируемом списке.

4.3.2. Реакция на ошибки

Модуль Errors определяет коды возможных ошибок и стандартный способ реакции на ошибки. Мы полагаем, что унифицированный механизм реакции на ошибку может существенно повысить удобство использования системы. Данный модуль определяет два необходимых базовых понятия: контекст ошибки (Error) и реакцию на ошибку (Exception).

Контекст ошибки - это объект, содержащий поле - код ошибки и метод, позволяющий получить в строке текст сообщения об ошибке.

```
TYPE
  Error = POINTER TO ErrorDesc;
  ErrorDesc = RECORD
    code*: LONGINT; -- код ошибки
    PROCEDURE (e: Error) perror(VAR msg: ARRAY OF CHAR);
    (* выдает в строку текст сообщения об ошибке *)
  END;
```

Тип Error может быть расширен добавлением новых полей, определяющих контекст ошибки. Так в модуле Files к контексту может быть добавлено имя файла, в загрузчике - имя модуля и версия его, и так далее.

Введение этого типа позволяет в широких пределах варьировать обработку ошибки. Могут быть использованы следующие варианты:

1) Трансляция ошибки

```
PROCEDURE DoIt(VAR res: Errors.Error);
BEGIN
  ...
  Try(res);
  IF res#NIL THEN RETURN END;
  ...
END DoIt;
```

Здесь и в следующих примерах:

```
PROCEDURE Try(VAR res: Errors.Error); ... END Try;
```

2) Реакция на код ошибки

```
PROCEDURE Find(....): BOOLEAN;
BEGIN
  Try(res);
  IF res=NIL THEN RETURN TRUE
  ELSIF res.code=Errors.no_entry THEN RETURN FALSE
  ELSE HALT(res.code); -- аварийное завершение команды
  END;
END Find;
```

3) Выдача сообщения об ошибке

```
Try(res);
IF res#NIL THEN
```

```
    res.perror(msg);
    print(msg);
  END;
```

4) Разбор контекста ошибки

```
  Try(res);
  IF res#NIL THEN
    WITH res: MyError DO ....
    | res: Files.Error DO ...
    ELSE HALT(res.code)
    END;
  END;
```

Таким образом мы можем различным способом обрабатывать ошибки, основываясь на одном универсальном механизме. Особенно важным является тот факт, что мы не теряем контекст ошибки при ее трансляции наверх. Так, например, если мы не смогли открыть окно, потому что не прочитался текст, потому что не загрузился элемент, потому что не загрузился модуль, потому что код модуля устарел, то напечатав сообщение об ошибке, мы узнаем причину и имя незагруженного модуля.

Второе понятие, определяемое этим модулем, это реакция на ошибку.

```
TYPE
  Exception = POINTER TO ExceptionDesc;
  ExceptionDesc = RECORD
    PROCEDURE (x: Exception) raise(e: Error);
  END;
```

Как правило, объект этого типа сопоставляется с некоторым другим объектом, например, файлом. В случае возникновения ошибки и после построения контекста будет вызван метод raise, который задает некоторую стандартную обработку ошибки. Такой обработкой может быть завершение команды, печать сообщения или возбуждение исключительной ситуации, определенной некоторой библиотекой.

При разработке некоторого приложения мы может произвольным образом задавать реакцию. Для некоторого простого приложения мы можем считать, что любая ошибка должна завершить команду. В более сложном приложении мы можем вести протокол ошибок, различать классы ошибок и так далее.

Приведем пример на базе модуля Files:

```
TYPE
  File = POINTER TO FileDesc;
  FileDesc = RECORD
    res*: Errors.Error;
    exc*: Errors.Exception;
    ....
  END;
  Error = RECORD (Errors.Error)
    name*: FileName;
  END;

PROCEDURE set_length(f: File; len: LONGINT);
  VAR e: Error; error_code: LONGINT;
BEGIN
  error_code:=try_set_length();
  IF error_code#0 THEN    -- не получилось установить длину
    NEW(e);
```

```
    e.name:=f.name;
    f.res:=e;
    IF f.exc=NIL THEN      -- реакция не определена
      HALT(e.code);        -- реакция по умолчанию
    ELSE
      f.exc.raise(e);      -- реакция, определенная пользователем
    END;
  END;
END set_length;
```

4.3.3. Каналы и объекты доступа

Данный раздел базируется на идеях, высказанных в работах [6, 34]. Основная идея заключается в разделении операций передачи данных и операций форматирования данных при реализации подсистемы ввода/вывода. Мы будем называть объекты, определяющие передачу данных, - каналами (файл, терминал, ...), а объекты, определяющие операции форматирования (структуру данных), - объектами доступа.

Такое разделение позволяет независимо расширять как каналы, так и объекты доступа и получить систему, в которой любой объект доступа может использоваться вместе с любым каналом. Мы поддерживаем схему N:1, то есть множество объектов доступа может быть связано с одним каналом, но каждый объект доступа связан с не более чем одним каналом. Объект доступа может взаимодействовать с каналом через операции, определенные в абстрактных классах. В некоторых случаях канал должен иметь возможность ассоциировать с объектом доступа некоторые

дополнительные атрибуты. Для этого вводиться дополнительный тип (Link), задающий связь между каналом и объектом доступа.

Тип Channel определяет (абстрактный) тип канала, тип Rider - абстрактный объект доступа. Метод attach присоединяет объект доступа к каналу.

Из описаний модуля Channels:

```
TYPE
  Channel = POINTER TO ChannelDesc;
  ChannelDesc = RECORD (Objects.ObjectDesc)
    res*: Errors.Error;
    exc*: Errors.Exception;

    PROCEDURE (c: Channel) attach(r: Rider);
      (* присоединение объекта доступа к каналу *)
    PROCEDURE (c: Channel) get_bytes(r: Rider;
                VAR x: ARRAY OF SYSTEM.BYTE; pos,len: LONGINT);
      (* чтение из канала неструктурированных данных *)
    PROCEDURE (c: Channel) put_bytes(r: Rider;
                VAR x: ARRAY OF SYSTEM.BYTE; pos,len: LONGINT);
      (* запись в канал неструктурированных данных *)
    PROCEDURE (c: Channel) get_pos(r: Rider): LONGINT;
      (* текущая позиция объекта доступа в канале *)
    PROCEDURE (c: Channel) set_pos(r: Rider; pos: LONGINT);
      (* изменение текущей позиции *)
END;

Link = POINTER TO LinkDesc;
LinkDesc = RECORD END;
```

```
Rider = POINTER TO RiderDesc;
RiderDesc = RECORD (Objects.ObjectDesc)
  res*: Errors.Error;
  exc*: Errors.Exception;
  base-: Channel;
  link-: Link;
  PROCEDURE (r: Rider) get(VAR x: SYSTEM.BYTE);
  PROCEDURE (r: Rider) get_int(VAR i: LONGINT);
  PROCEDURE (r: Rider) get_real(VAR x: REAL);
  PROCEDURE (r: Rider) get_str(VAR s: ARRAY OF CHAR);
  ...
  PROCEDURE (r: Rider) put(x: SYSTEM.BYTE);
  PROCEDURE (r: Rider) put_int(x: LONGINT);
  PROCEDURE (r: Rider) put_real(x: REAL);
  PROCEDURE (r: Rider) put_str(s: ARRAY OF CHAR);
  ...
END;
```

Модуль XRiders определяет стандартный объект доступа, который читает и пишет данные в переносимом бинарном формате.

Тип Files.File расширяет тип Channel (см. 4.3.6).

4.3.4. Загрузчик

Модуль Loader реализует операции загрузки/выгрузки модуля, динамический вызов команды и динамического выделения памяти. Операция динамического выделения памяти, то есть выделения памяти с указанием типа записи в виде пары (ИмяМодуля, ИмяТипа) отсутствует в системе Оберон и была предложена в работе [36].

Из описаний модуля Loader:

```
TYPE
  Module = POINTER TO ModuleDesc;
  ModuleDesc = RECORD (Objects.ObjectDesc)
  END;

PROCEDURE lookup(name: ARRAY OF CHAR; VAR m: Module;
                                        VAR res: Errors.Error);
(* Возвращает по имени модуль. Если модуль не загружен,
   то пытается загрузить его.
*)

PROCEDURE unload(name: ARRAY OF CHAR; VAR res: Errors.Error);
(* Выгружает модуль *)

PROCEDURE call(m: Module; cmd: ARRAY OF CHAR;
                      VAR res: Errors.Error);
(* Вызов команды *)

PROCEDURE new(m: Module; type: ARRAY OF CHAR;
                         VAR res: Errors.Error): Objects.Object;
(* Выделение памяти *)
```

Кроме того, модуль реализует итераторы, позволяющие получить список всех загруженных модулей, список команд, список имен типов записей модуля и процедуру разбора имени команды на составные части:

```
PROCEDURE parse(command: ARRAY OF CHAR; VAR e: Entry);
```

Здесь тип Entry определен как:

```
TYPE
  Entry = RECORD
    module: Name;
    entry : Name;
    new : BOOLEAN;
    res : Errors.Error;
  END;
```

Строка command может быть представлена как

```
  ИмяМодуля "." ИмяКоманды
```

или

```
  ИмяМодуля "." "^" ИмяТипа
```

Во втором случае, строка обозначает операцию динамического выделения памяти для указателя на тип, задаваемые парой (ИмяМодуля, ИмяТипа). Поле new при этом выставляется в TRUE.

Данный модуль является системно-зависимым. Версии модуля для версий Mithril/Excelsior и Mithril/MS-DOS реализованы А. Хапугиным.

4.3.5. Постоянные объекты

Постоянными объектами мы будем называть объекты, для которых определены операции записи и восстановления. В системе Оберон понятие постоянного объекта не выделено. Поэтому такие объекты определяются в каждом конкретном случае (например, Элементы в редакторе Write [26] или графические объекты в редакторе Condor [31]). Мы полагаем, что понятие постоянного

объекта позволяет унифицировать и упростить разработку различных приложений.

Постоянные объекты определяются в модуле Storage. Для такого объекта определены методы записи и восстановления атрибутов объекта, метод инициализации и метод, выдающий имя команды создания объекта.

Из описаний модуля Storage:

```
TYPE
  Object = POINTER TO ObjectDesc;
  ObjectDesc = RECORD (Objects.ObjectDesc)
    PROCEDURE (o: Object) externalize(r: XRiders.Rider);
      (* записывает атрибуты объекта в объект доступа *)
    PROCEDURE (o: Object) allocator(VAR s: ARRAY OF CHAR);
      (* возвращает строку команды создания объекта *)
    PROCEDURE (o: Object) constructor;
      (* инициализирует объект *)
    PROCEDURE (o: Object) internalize(r: XRiders.Rider);
      (* читает атрибуты объекта из объекта доступа *)
  END;
```

Кроме описания постоянного объекта, модуль определяет процедуры чтения и записи постоянного объекта. Образ постоянного объекта в файле выглядит следующим образом:

```
  OBJ_TAG команда_создания_объекта длина атрибуты_объекта
```

или

```
  NIL_TAG
```

если записывается объект со значением NIL.

Процедура put_object записывает произвольный объект в объект доступа. Процедура get_object - восстанавливает объект.

```
PROCEDURE put_object(r: XRiders.Rider; o: Object);
PROCEDURE get_object(r: XRiders.Rider; VAR o: Object;
                                       VAR res: Errors.Error);
```

Если модуль, имя которого определено в команде, не удалось загрузить, то процедура пропускает атрибуты объекта и возвращает ошибку. Дополнительная процедура get_header используется в тех случаях, когда пропускать атрибуты объекта не надо (например, при чтении элемента в тексте; см. 4.3.8).

4.3.6. Файловая система

Модуль Files определяет тип File, являющийся расширением типа Channels.Channel и набор процедур открытия, создания, закрытия файлов, доступа к атрибутам файла и итерации директории. Заметим, что операции чтения и записи определены только для объекта доступа. К файлу можно присоединить произвольный объект доступа, при этом методы attach и set_pos (см. 4.3.3) присоединяют специальный объект связи (Link) к объекту доступа.

Из описаний модуля Files:

```
  File = POINTER TO FileDesc;
  FileDesc = RECORD (Channels.ChannelDesc)
  END;
  Link = POINTER TO LinkDesc;
  LinkDesc- = RECORD (Channels.LinkDesc)
```

```
    iolen-: LONGINT; (* длина последней операции *)
    fpos-: LONGINT; (* позиция в файле *)
    eof-: BOOLEAN; (* признак конца файла *)
  END;

PROCEDURE create(dir: File; path,mode: ARRAY OF CHAR): File;
PROCEDURE open(dir: File; path,mode: ARRAY OF CHAR): File;
(* Если dir=NIL, то файл открывается/создается
   на текущей директории.
*)

PROCEDURE close(f: File);
(* Закрытие файла *)
```

Расширение типа File не поддерживается, хотя и не запрещается. Все операции над файлом определены как статические процедуры. Приведем пример, чтения массива целых чисел из файла.

```
PROCEDURE read(path: ARRAY OF CHAR; VAR ints: ARRAY OF LONGINT);
  VAR f: Files.File; R: XRiders.Rider; i: LONGINT;
BEGIN
  f:=Files.open(NIL,path,"r");
  IF f.res#NIL THEN (* реакция на ошибку *) END;
  NEW(R);
  f.attach(R);            -- присоединили объект доступа к файлу
  FOR i:=0 TO LEN(ints)-1 DO
    R.get_longint(ints[i])
  END;
  Files.close(f);
END read;
```

По умолчанию для файла устанавливается реакция на ошибку (Exception, см. 4.3.2), которая вызывает прекращение выполнения команды для всех ошибок, кроме ошибок открытия и создания файла.

Для итерации директории определяется специальный тип итератора. Процедура iter_dir вызывает метод entry для каждого файла в директории.

```
TYPE
  Iter = RECORD
    PROCEDURE (VAR i: Iter) entry(name: ARRAY OF CHAR;
                                  class: SET);
  END;

PROCEDURE iter_dir(dir: File; VAR iter: Iter);
(* Итератор директории; вызывает метод iter.entry для
   каждого файла в директории.
*)
```

Такой механизм позволяет удобным образом обойти директорию и построить необходимую для приложения структуру. Например, для построения линейного списка некоторое приложение может определить следующее расширения типа итератора:

```
TYPE
  Node = POINTER TO NodeDesc;
  NodeDesc = RECORD
    name : Name;
    class: SET;
    next : Node;
  END;
  IterList = RECORD (Files.Iter)
    head: Node;
```

```
  END;

PROCEDURE (i: Iter) entry(name: ARRAY OF CHAR; class: SET);
  VAR n: Node;
BEGIN
  NEW(n); COPY(name,n.name); n.class:=class;
  n.next:=i.head; i.head:=n; -- ввязали в линейный список
END entry;
```

На этом примере видно важное отличие использования в итерации объекта вместо процедурного значения. Кроме действия (метода) объект содержит некоторый контекст. При использовании процедуры нам бы пришлось определять этот контекст (в данном случае переменную head) на глобальном уровне.

Данный модуль является системно-зависимым. Для систем Mithril/Kronos и Mithril/MS-DOS он реализован на базе библиотеки BIO, которая является стандартной библиотекой ОС Excelsior и была перенесена в среду MS-DOS при переносе компилятора.

4.3.7. Фонты

Модуль Fonts определяет абстрактный тип Font. Каждый фонт имеет атрибуты: высоту, максимальную ширину символа и некоторые другие. Конкретный фонт определяется как расширение абстрактного. Каждый конкретный фонт должен реализовать два метода: char_attr и width, которые возвращают атрибуты символа. Каждый драйвер экрана работает с некоторым подмножеством конкретных фонтов.

```
TYPE
  Font = POINTER TO FontDesc;
  FontDesc = RECORD (Storage.ObjectDesc)
    h: INTEGER; -- высота фонта
    w: INTEGER; -- максимальная ширина символа
    name: Objects.String; -- имя фонта
    PROCEDURE (f: Font) char_attr(ch: CHAR;
                                  VAR dx,x,y,w,h: INTEGER);
    (* возвращает атрибуты символа *)
    PROCEDURE (f: Font) width(ch: CHAR): INTEGER;
    (* возвращает ширину символа *)
  END;
```

Процедура this реализует поиск фонта по имени. Все загруженные фонты связаны в нетрассируемый список (см 4.3.1). Неиспользуемый фонт будет удален из списка при выполнении операции сборки мусора. Если фонт не загружен, то процедура this пытается его загрузить, вернее прочитать его из файла с именем, построенным стандартным образом по имени фонта. Приведем схему реализации этой процедуры:

```
VAR fonts: Closure.Trailer; -- нетрассируемый список всех фонтов

PROCEDURE this(name: ARRAY OF CHAR): Font;
  VAR l: Closure.Link; f: Font;
BEGIN
  l:=fonts.link;
  WHILE l#NIL DO
    IF l.obj(Font).name=name THEN RETURN l.obj(Font) END;
    l:=l.next;
```

```
  END;
  RETURN read_font(name)
END this;

PROCEDURE read_font(name: ARRAY OF CHAR): Font;
  VAR f: Files.File; R: XRiders.Rider; o: Objects.Object;
BEGIN
  f:=Files.open(NIL,name,R);
  NEW(R); f.attach(R);
  Storage.get_object(R,o);      -- читаем объект из файла
  WITH o: Font DO               -- проверяем, что прочитали фонт
    Closure.insert(fonts,o);    -- присоединяем его к списку
    RETURN o
  END;
END read_font;
```

Так как фонт читается из файла, как объект, то после чтения мы получаем некоторое расширение (конкретизацию) фонта. Система может одновременно использовать разные виды фонтов с различным представлением. Для подключения нового фонта к системе достаточно оформить его в виде объекта в стандарте модуля Storage.

Специальный механизм реализован в данном модуле для поддержки так называемых "сессий". Рассмотрим операцию записи текста, в котором используется много различных фонтов. Для каждого фрагмента текста нам необходимо указать фонт, который для него используется. Так как размер данных для фонта достаточно велик, то для указания фонта мы будем записывать его имя. Если один фонт используется для различных фрагментов, то

объем записываемой информации можно существенно уменьшить, если пронумеровать все фонты, используемые в данном тексте, и записывать имя фонта только при первом использовании, а при следующих указывать только его локальный номер. Для ведения такой локальной нумерации мы использует метод, аналогичный методу, описанному в [36].

Кроме уменьшения объема информации сессия ускоряет чтение текста, так как повторное обращение к некоторому фонту выполняется за константное время (не содержит поиска).

4.3.8. Тексты

Модуль Texts определяет понятие текста, близкое к соответствующему понятию системы Оберон, но с двумя существенными изменениями:

- использование методов вместо статических процедур упрощает возможность дальнейшего расширения;
- элементы (абстрактные литеры) реализованы на базовом уровне.

Понятие текста является очень важным, так как практически все, с чем мы имеем дело, является текстом. Мы постараемся подробно описать логическое устройство текста и операции над ним.

Система Мифрил (вслед за системой Оберон) поддерживает достаточно сложное понятие текста. Каждая литера в тексте обладает своим набором атрибутов, а именно цветом, вертикальным смещением относительно базовой линии и принадлежит к некоторому фонту.

Кроме типа Text модуль определяет тип Elem (абстрактную литеру), объекты доступа (чтения - Reader, Scanner и записи - Writer) и объект (Buffer), позволяющий работать с фрагментом текста, как с атомарным объектом. Структура текста скрыта в данном модуле.

Из описаний модуля Texts:

```
  Buffer = POINTER TO BufferDesc;
  BufferDesc = RECORD (Objects.ObjectDesc)
    len-: LONGINT;              -- текущая длина

    PROCEDURE (b: Buffer) append(a: Buffer);
      (* добавление буфера *)
    PROCEDURE (b: Buffer) copy(a: Buffer);
      (* копирование буфера *)
    PROCEDURE (b: Buffer) open;
      (* инициализцаия (пустого) буфера *)
  END;
```

Все операции над буфером трактуют буфер, как атомарный объект, то есть не позволяют работать с его частями. Буфер хранит некоторый фрагмент текста с атрибутами.

```
  Text = POINTER TO TextDesc;
  TextDesc = RECORD (Storage.ObjectDesc)
    len-: LONGINT;              -- длина текста

    PROCEDURE (T: Text) open;
      (* инициализация (пустого) текста *)
    PROCEDURE (T: Text) append(a: Buffer);
      (* добавление буфера к концу текста *)
```

```
    PROCEDURE (T: Text) insert(pos: LONGINT; i: Buffer);
      (* вставка буфера в позицию pos *)
    PROCEDURE (T: Text) delete(beg,end: LONGINT);
      (* удаление фрагмента текста [beg..end[ *)
    PROCEDURE (T: Text) change_looks(beg,end: LONGINT; sel: SET;
                    font: Fonts.Font; col: SET; voff: INTEGER);
      (* изменение атрибутов фрагмента текста *)
    PROCEDURE (T: Text) save(pos,end: LONGINT; b: Buffer);
      (* копирование фрагмента текста в буфер *)
  END;
```

Текст - это массив литер с номерами в диапазоне [0..len-1]. Операции над текстом позволяют вставить буфер в указанную позицию, удалить фрагмент текста, изменить атрибуты фрагмента и скопировать фрагмент текста (с атрибутами) в буфер. Внутреннее представление текста реализовано в виде двусвязного списка фрагментов текста с одинаковыми атрибутами, что позволяет достаточно быстро выполнять все операции над текстом.

Важно заметить, что текст является постоянным объектом, то есть его можно записать в файл и прочитать, как целое.

Каждый текст (или фрагмент) может быть изображен различным образом в разных объектах (например, окнах). При изменении текста (вставка, удаление, изменение атрибутов) модуль посылает сообщения об изменении всем объектам, ассоциированным с данным текстом. Для этого используется механизм нотификации. Каждые объект, изображающий данный текст, должен зарегистрироваться в тексте. Специальный объект используется для посылки сообщения.

```
  Noticer = POINTER TO NoticerDesc;
  NoticerDesc = RECORD (Objects.ObjectDesc)
```

```
    PROCEDURE (n: Noticer) notify(T: Text; op: INTEGER;
                                  beg,end: LONGINT);
      (* сообщение об изменении текста в диапазоне [beg..end[;
         op принимает значения insert, delete, replace.
      *)
  END;
```

Текст поддерживает (нетрассируемый) список таких объектов. Введение дополнительного объекта позволяет достичь независимости текста от изображающих его объектов. Здесь мы используем тот же прием, что и при разделении каналов и объектов доступа (см. 4.3.3).

Рассмотрим далее объекты доступа к тексту:

```
  Reader = POINTER TO ReaderDesc;
  ReaderDesc = RECORD (Objects.ObjectDesc)
    text-: Text;
    pos-: LONGINT;          -- текущая позиция в тексте
    eot*: BOOLEAN;          -- признак конца текста
    col*: SET;              -- цвет последней прочитанной литеры
    voff*: INTEGER;         -- вертикальное смещение
    font*: Fonts.Font;      -- фонт
    elem*: Elem;

    PROCEDURE (r: Reader) open(T: Text; pos: LONGINT);
      (* устанавливает объект чтения в позицию pos *)
    PROCEDURE (r: Reader) read(VAR ch: CHAR);
      (* читает очередной символ из текста *)
  END;
```

Объект чтения Reader позволяет прочитать из текста литеру и узнать ее атрибуты (поля col, voff, font). Если прочитанная литера является элементом, то соответствующий объект выставляется в поле elem. После чтения литеры текущая позиция увеличивается.

Дополнительный объект чтения (Scanner) выполняет при чтении лексический анализ и позволяет получить число (записанное в одном из принятых в Модуле-2 и Обероне-2 формате), строку, идентификатор или специальную литеру.

```
  Writer = POINTER TO WriterDesc;
  WriterDesc = RECORD (Objects.ObjectDesc)
    buf-: Buffer;
    col*: SET;
    voff*: INTEGER;
    font*: Fonts.Font;
    PROCEDURE (w: Writer) open;
    PROCEDURE (w: Writer) element(e: Elem);
    PROCEDURE (w: Writer) write(ch: CHAR);
    PROCEDURE (w: Writer) print(fmt: ARRAY OF CHAR;
                                    SEQ x: BYTE);
    PROCEDURE (w: Writer) set_color(col: SET);
    PROCEDURE (w: Writer) set_font(f: Fonts.Font);
    PROCEDURE (w: Writer) set_offset(voff: INTEGER);
  END;
```

Объект записи позволяет добавить к буферу литеру, элемент или последовательность литер. Процедуры set_* позволяют установить атрибуты литер.

Если W - объект записи, а T - текст, то последовательность вызовов:

```
W.open;                 -- инициализация пустого объекта записи
W.set_color(red);       -- изменили цвет
W.print('Hello, ');
W.set_color(yellow);    -- изменили цвет
W.print('world!');
T.insert(10,W.buf);
```

вставит в текст в позицию 10 предложение "Hello, world!", причем первое слово будет изображено красным, а второе – желтым цветом. После вставки операция insert вызовет сообщение нотификации

```
n.notify(T,insert,10,24)
```

для всех объектов, присоединенных к тексту T (то есть для всех объектов, изображающих этот текст).

Осталось рассказать об абстрактных литерах (элементах):

```
Elem = POINTER TO ElemDesc;
ElemDesc = RECORD (Storage.ObjectDesc)
  dx,w,h: INTEGER;
  alien-: Objects.String;
  text-: Text;
  PROCEDURE (e: Elem) copy(VAR x: Elem);
    (* копирование элемента *)
  PROCEDURE (e: Elem) handle(VAR M: Objects.Message);
    (* управление элементом *)
END;
```

Элемент является постоянным объектом и обладает геометрическими размерами. Строка alien содержит текст команды создания элемента, если при чтении текста не удалось загрузить модуль, реализующий данный элемент. При записи такого "чужого" элемента

будут скопированы атрибуты объекта, что позволяет при восстановлении модуля, реализующего объект, восстановить изображение и поведение элемента. Набор методов для изображения элементов определяется в модуле Elements. В целом реализация элементов близка к описанной в [26] (см. также 3.3).

Кроме описаний типов и методов модуль также реализует набор статических процедур. Кратко упомянем некоторые из них.

```
PROCEDURE attach(T: Text; n: Noticer);
(* Добавляет к тексту объект нотификации. *)

PROCEDURE broadcast(T: Text; pos,end: LONGINT;
                                VAR m: Objects.Message);
(* Посылает сообщение всем элементам в диапазоне [beg..end[. *)

PROCEDURE open(T: Text; fname: ARRAY OF CHAR);
(* Открывает текст, хранящийся в файле с именем fname.
   Если этот файл не содержит информации об атрибутах,
   то считается, что все литеры текста имеют некоторые
   стандартные атрибуты.
*)

PROCEDURE store(T: Text; f: Files.File; pos: LONGINT;
                                    VAR len: LONGINT);
(* Записывает текст в файл, начиная с позиции pos. *)
```

4.3.9. Оконная система

Оконная (под)система в системе Мифрил реализована А. Никитиным. Оконная система достаточно подробно описана в

работе [37]. В данном разделе мы перечислим только основные особенности системы.

Ядро подсистемы состоит из двух модулей: Windows и Screens. Модуль Screens определяет тип Screen - абстрактный драйвер экрана. Расширение (конкретизация) экрана реализует операции рисования графических примитивов, изменения палитры и работы с курсором для конкретного вида графического устройства. Система позволяет одновременно работать на нескольких физических экранах. Каждому экрану соответствует некоторое окно. Все окна-экраны расположены на мета-окне.

Модуль Windows определяет базовый тип окна (Window). Каждое окно может содержать подокна. Иерархия окон ничем не ограничена. Каждое окно имеет имя, и модуль определяет операции доступа к окну по пути, аналогично поиску файла в иерархической файловой системе. Окно является постоянным объектом, при записи окна записываются и все его подокна. Механизм именования позволяет исключить необходимость в явных ссылках на подокно и обеспечить восстановление всей иерархии окон при чтении из файла.

Каждое окно содержит ссылку на драйвер экрана, и все операции рисования выполняются через примитивы конкретного драйвера экрана. Модуль поддерживает для каждого окна список видимых блоков (прямоугольников) данного окна. Все графические примитивы выполняются по списку видимых блоков.

4.3.10. Инициализация и основной цикл системы

В предыдущих пунктах (4.3.1-4.3.9) мы описали основные понятия системы. Модуль Kernel создает среду исполнения и интерфейс пользователя на базе этих понятий. Запуск системы

Мифрил происходит при запуске данного модуля. При инициализации модуля происходит создание драйвера первичного экрана и создание окна, соответствующему этому стандартному экрану. После этого модуль выполняет набор команд, задающих конфигурацию системы. Стандартная (минимальная) конфигурация состоит из двух окон: окна стандартного вывода (System.Log) и окна, содержащего набор команд (System.Tool). Заметим, что оба этих окна являются текстовыми (см. 4.4.2) расширениями Windows.Window и реализованы выше уровня ядра.

После выполнения инициализации модуль Kernel начинает выполнять так называемый основной цикл. В основном цикле система ожидает некоторого внешнего события (клавиатура, мышь, сеть, ...), по наступлению такого события определяет объект, который должен реагировать на это событие, и запускает его обработку.

При изменении состояния мыши обработчиком события (как правило) является окно, на которое указывает мышь, нажатие на кнопку клавиатуры обрабатывается некоторым текущим (фокусным) окном. Обработка события может заключаться просто в передаче этого события некоторому другому (под)окну или объекту (например, элементу в текстовом окне).

4.4. Оболочка системы

Оболочка системы содержит набор расширений базовых понятий, необходимых для реализации пользовательского интерфейса и набор необходимых команд. Необходимо отметить, что на базе ядра системы можно построить совершенно различные оболочки. Более того, оболочка системы реализована таким образом, что любой ее

объект может быть изменен или расширен и это не повлияет на работоспособность остальных объектов.

Оболочка системы Мифрил быстро развивается. В следующих пунктах мы опишем некоторые важные ее части, не претендуя на полноту описания. Внимательный читатель отметит существенное отличие данного раздела от предыдущего. Если при описании ядра системы нас интересовали вопросы методологии проектирования и выбора правильного интерфейса, то в данном разделе мы просто описываем свойства полученной системы.

4.4.1. Поддержка разработки пользовательского интерфейса

Все многооконные системы используют более-менее стандартный набор объектов пользовательского интерфейса: кнопки, меню, позиционеры, диалоговые окна и так далее. Все эти объекты реализованы и в системе Мифрил. Каждый такой объект является расширением специального расширения окна (селектора). Универсальная оконная система позволяет разместить на окне различные подокна и указать, как размеры и местоположение подокон должны меняться при изменении размеров окна. Механизм именования окон позволяет найти нужное подокно по имени, независимо от его расположения, и послать ему некоторое сообщение.

В настоящее время разрабатывается редактор интерфейсов, которые позволит собрать нужную конфигурацию окон в диалоге с пользователем, то есть разместить кнопки, текст, и так далее на некотором несущем окне (панели). Так как окно является постоянным объектом, то приготовленную панель можно записать в файл.

Каждый селектор содержит специальный объект (актор), и все сообщения, полученные селектором, транслируются этому объекту. Таким образом, вся неинтерфейсная часть приложения может быть реализована набором акторов. Актор также является постоянным объектом, поэтому конструируя интерфейс, мы можем указать команды создания нужных нам акторов. Если часть акторов не реализована, то при чтении панели такие акторы будут заменены на некоторые пустые, и мы можем получить частично реализованное приложение. Для полной реализации приложения нам нужно реализовать все акторы.

В такой модели приложение представлено в виде файла, который хранит образ панели и набора модулей, реализующих различные акторы. Более того, если пользователю не нравится интерфейс приложения, то он может отредактировать образ панели и таким образом настроить приложение. При чтении панели мы можем указать, что панель должна быть создана не на экране, а на некоторой другой панели, что позволяет нам создавать библиотеки панелей.

Исследования показывают, что для различных приложений часть приложения, реализующего пользовательский интерфейс, может составлять от 29 до 88% (см. например [38]). Поэтому возможность отделить приложение (в виде набора акторов) от интерфейса является очень важной и позволяет резко ускорить разработку приложений.

Кроме селекторов, оболочка реализует стандартную панель, состоящую из заголовка, меню, рабочего окна и рамки, рабочие органы которой позволяют изменять размеры и местоположение окна. Заголовок, меню и рабочее окно являются окнами и могут быть

установлены после создания панели. Как правило, меню и рабочее окно являются текстовыми окнами (см. 4.4.2).

4.4.2. Текстовые окна

Мы уже несколько раз упоминали особую роль текстовых окон в системе. Текстовые окна реализуются модулем TWindows. Текстовые окна изображают некоторый текст, возможно, содержащий элементы. Для изображения элемента текстовое окно посылает элементу сообщения, определенные в модуле Elements. Рисование строки выполняется в два прохода по тексту. На первом проходе вычисляется высота и ширина строки, при этом элементу посылается сообщение "приготовься к рисованию", и элемент может изменить свои размеры. На втором проходе обычные символы рисуются, и каждому элементу посылается сообщение "нарисуй себя".

Основные операции над текстом, изображенным в текстовом окне, выполняются при помощи мыши: установка курсора, удаление фрагмента текста, копирование фрагмента текста, изменение атрибутов фрагмента и запуск команды. Специальный утилитный модуль реализует дополнительные команды, такие как открытие окна, запись текста, поиск подстроки, и так далее. Вертикальная полоса в левой части текстового окна выделена под операции позиционирования в тексте.

Мы полагаем, что для подготовки документации должно использоваться специальное расширение текстового окна, выполняющее операции форматирования текста и другие специфичные операции.

4.4.3. Текстовые элементы

Как показано в 3.3, текстовые элементы могут быть расширены различными способами, существенно увеличивая возможности редактора. В системе Мифрил в настоящее время реализованы два расширения элементов - элементы пометки и командные элементы.

Элементы пометки используются для вставки в текст некоторых пометок, например, сообщений об ошибках компилятора. Каждая пометка может быть в двух состояниях - открытом и закрытом. В закрытом состоянии пометка изображается небольшим квадратом, а в открытом - прямоугольником с некоторым текстом. Переключение состояния выполняется с помощью средней кнопки мыши. Кроме того, с помощью мыши можно перейти на предыдущую и следующую пометку. Эти элементы являются временными, то есть при записи в файл они игнорируются.

Командные элементы также могут быть в двух состояниях. В закрытом состоянии элемент изображается некоторой иконкой, в открытом - прямоугольником, содержащим текст команды. В любом состоянии можно запустить команду, нажав на среднюю кнопку мыши. Такие элементы позволяют формировать строки меню из иконок, что очень важно на экранах маленьких размеров, так как текст команды занимает существенно больше места. В системе определен достаточно большой набор иконок, что позволяет при создании командного элемента выбрать подходящий иероглиф для данного действия. Если пользователь не понимает смысла иконки, то он может раскрыть элемент и прочитать текст команды. Специальный редактор иконок является частью системы и позволяет удобным образом определить новую иконку.

В дальнейшем мы планируем реализовать большую часть элементов, описанных в [26], в первую очередь элементы, поддерживающие свойства гипертекста.

4.4.4. Основные наборы команд

Система включает три основных набора команд: системные команды, команды редактора и компилятора. Системный набор команд включает команды открытия текстовых окон, выдачу списка загруженных модулей и списка команд в данном модуле, выгрузку модулей, запуск сборки мусора и так далее. Набор команд редактора содержит команды изменения атрибутов, поиска подстроки, записи текста. Набор команд компилятора содержит команду вызова компилятора.

Команда создания текстового окна создает строку меню стандартного вида. Эта строка содержит элементы-команды для самых распространенных команд, а именно команды копирования и закрытия окна, записи текста и поиска подстроки.

Каждая команда может иметь аргументы. Доступ к аргументам команды полностью аналогичен доступу, принятому в системе Оберон.

4.5. Редактор формул, как пример разработки приложения

В данном пункте мы опишем структуру некоторого простого редактора математических формул. Мы будем представлять формулу в виде элемента в тексте. Каждая формула состоит из литер различных алфавитов и специальных символов, таких как корень, интеграл, сумма, деление и так далее. Базовый уровень текста в

системе позволяет нам получить текст, состоящий из литер различных алфавитов. Для этого достаточно иметь набор нужных фонтов (например, фонт с греческим алфавитом). Следовательно, нам достаточно определить изображения специальных символов.

Следующий важный вопрос - это способ задания формулы. Мы полагаем, что наиболее удобным является представление формулы на некотором простом функциональном языке, например:

```
        div("a+b","c+d")

                              a+b
должно быть отображено в      ---.
                              c+d
```

В текст на таком языке достаточно удобно вносить правки, и мы избавлены от необходимости редактировать формулы в их конечном представлении. Добавление символических имен для частей формулы позволит нам существенно упростить задание формул.

Перейдем теперь к реализации такого редактора. С точки зрения текста формула является элементом. Элемент содержит объект (модель), описывающий внутреннюю структуру формулы. Формула состоит из различных объектов, являющихся расширениями некоторого абстрактного объекта, назовем его частью (Piece). Такой объект является постоянным объектом (мы должны уметь хранить формулы в файле), и над ним должна быть определена операция рисования и операция подготовки к рисованию (далее мы объясним это подробнее).

```
TYPE
  Piece = POINTER TO PieceDesc;
  PieceDesc = RECORD (Storage.Object)
    w,h: INTEGER;
```

```
    PROCEDURE (p: Piece) prepare;
    PROCEDURE (p: Piece) draw(v: Windows.Window; x,y: INTEGER);
  END;
```

Очевидным расширением такого абстрактного объекта, является объект, содержащий некоторый текст.

```
TYPE
  TextPiece = POINTER TO TextPieceDesc;
  TextPieceDesc = RECORD (PieceDesc)
    text: Texts.Text;
  END;
```

При выполнении операции prepare объект вычисляет размеры объемлющего прямоугольника, а операция draw изображает текст в прямоугольнике, левый нижний угол которого задан точкой (x,y). В описании формулы текстовая часть задается строкой, например "a+b". Важно заметить, что при создании такого объекта символы строки копируются вместе со своими атрибутами.

Рассмотрим теперь определение специального символа, на примере операции деления, изображаемой горизонтальной чертой.

```
TYPE
  Div = POINTER TO DivDesc;
  DivDesc = RECORD (PieceDesc)
    up: Piece; -- верхняя часть (над дробной чертой)
    dw: Piece; -- нижняя часть (под дробной чертой)
  END;
```

Реализуем методы подготовки и рисования:

```
PROCEDURE (d: Div) prepare;
```

```
BEGIN
  d.up.prepare;              -- вычислили размеры верхней части
  d.dw.prepare;              -- вычислили размеры нижней части
  d.w:=max(d.up.w,d.dw.w);   -- ширина деления равна ширине
                             -- большей части
  d.h:=d.up.h+d.dw.h+dH;     -- высота деления равна сумме высот
                             -- плюс некоторая константа
END prepare;

PROCEDURE (d: Div) draw(v: Windows.Window; x,y: INTEGER);
  VAR Y: INTEGER;
BEGIN
  d.dw.draw(v,x+(d.w-d.dw.w) DIV 2,y);
  -- нарисовали нижнюю часть
  d.up.draw(v,x+(d.w-d.up.w) DIV 2,y+d.dw.h+dH);
  -- нарисовали верхнюю часть
  Y:=y+d.dw.h;
  wnd.line(v,x,Y,x+d.w-1,Y);
  -- нарисовали черту между частями
END draw;
```

Эти две процедуры полностью реализуют рисование деления, разве что в настоящем редакторе формул желательно вычислять ширину разделяющей линии в зависимости от высоты частей. Определив деление, мы можем строить произвольные формулы, состоящие из делений, например: div("a+b",div("c+d","x*y")). Также просто реализуются операции записи и чтения объекта.

```
PROCEDURE (d: Div) externalize(r: XRiders.Rider);
BEGIN
```

```
  Storage.put_object(d.up);
  Storage.put_object(d.dw);
END externalize;

PROCEDURE (d: Div) internalize(r: XRiders.Rider);
  VAR o: Objects.Object; res: Errors.Error;
BEGIN
  Storage.get_object(r,o,res); IF res#NIL THEN ... END;
  d.up:=o(Piece);
  Storage.get_object(r,o,res); IF res#NIL THEN ... END;
  d.dw:=o(Piece);
END internalize;
```

Мы опускаем детали, связанные с реакцией на ошибку при чтении объекта.

Точно также мы можем определить и более сложные конструкции. Например, интеграл будет включать три части: подынтегральное выражение и две необязательные части, определяющие граничные выражения.

До сих пор открытым оставался вопрос об отображении записи формулы в ее представление в виде дерева частей. Описание формулы является вызовом функции, имя которой задается в виде команды (M.P). Если имя модуля отсутствует, то подразумевается имя некоторого стандартного модуля. Операндами функции могут быть также вызовы функции, а также строки текста и числа. Процедур разбора описания строит для каждой функции список операндов следующим образом:

- строка : создается объект TextPiece и вставляется в список операндов;

- число : вставляется в список операндов;
- функция: вызывается команда, определенная именем функции; эта команда должна создать объект, являющийся расширением Piece; после чего вызывается метод инициализации объекта с параметром - списком аргументов. Готовый объект вставляется в список операндов.

Описывая тип Piece (см. выше) мы опустили метод инициализации объекта. Полное описание этого типа выглядит так:

```
TYPE
  Piece = POINTER TO PieceDesc;
  PieceDesc = RECORD (Storage.Object)
    w,h: INTEGER;

    PROCEDURE (p: Piece) prepare;
    PROCEDURE (p: Piece) draw(v: Windows.Window; x,y: INTEGER);
    PROCEDURE (p: Piece) init(n: Node);
  END;
```

Здесь тип Node определен как:

```
  Node = POINTER TO NodeDesc;
  NodeDesc = RECORD
    class: (piece,int,real);
    piece: Piece;
    int : LONGINT;
    real : LONGREAL;
    next : Node;
  END;
```

При разборе описания: div("a+b","c+d") будет построен список из двух узлов, содержащих части, соответствующие текстам "a+b" и "c+d", и этот список будет передан методу инициализации объекта типа "Div".

```
PROCEDURE (d: Div) init(n: Node);
BEGIN
  IF n=NIL THEN error("мало аргументов"); RETURN END;
  IF n.class#piece THEN error("не тот аргумент"); RETURN END;
  d.up:=n;
  n:=n.next;
  IF n=NIL THEN error("мало аргументов"); RETURN END;
  IF n.class#piece THEN error("не тот аргумент"); RETURN END;
  IF n.next#NIL THEN error("лишний аргумент") END;
END init;
```

Каждый объект, получая список аргументов, проверяет типы и количество аргументов и выполняет свою инициализацию. Процедура разбора не знает о расширениях типа Piece, за исключением текстового расширения. Таким образом, при добавлении нового расширения не требуется изменения или перекомпиляции ядра редактора формул.

Головной модуль редактора формул реализует единственную команду "Compile", которая разбирает описание формулы, создает элемент и вставляет его в позицию курсора. Методы элемента транслируются в методы корневой части формулы. Ядро редактора, включающее текстовые части и процедуру разбора, состоит из 300 строк (!) на языке Оберон-2.

4.6. Методы переноса системы

В настоящее время выполнен перенос системы в среду MS-DOS для старших моделей PC. Для переноса использовались переносимые компиляторы с генерацией для i386/486 [24].

Основной сложностью была реализация специальной утилиты – загрузчика (DOS extender), который переводит процессор в защищенный режим и обеспечивает доступ к базовым функциям операционной системы. Реализация такого загрузчика была выполнена О. Шатохиным.

После реализации загрузчика, перенос сводился к реализации модуля динамической поддержки и набора конкретных драйверов, а именно:

- драйвера экрана. Использовалась графическая библиотека, перенесенная с OS Excelsior;
- драйвера фонтов;
- драйвера ввода (мышь, клавиатура)ж
- файловой системы.

Не считая графической библиотеки, которая была перенесена независимо, при переносе пришлось написать 1400 строк на Модуле-2 (модуль динамической поддержки) и около 2000 строк на Обероне-2 (динамический загрузчик, файловая система, драйверы).

Перенос первой версии системы был выполнен за неделю. Полученный при переносе опыт позволил существенно улучшить переносимость системы и обеспечить полную совместимость приложений на уровне исходных текстов.

4.7. Выводы

Система Оберон является первым примером переносимой расширяемой системы. Разработка системы Мифрил проводилась на основе анализа достоинств и недостатков системы Оберон. С нашей точки зрения, основными достоинствами этой системы (унаследованными системой Мифрил) являются:

- отказ от жесткой, монолитной структуры; организация системы на базе динамической загрузки и динамического запуска команд, и тем самым поддержка динамической расширяемости;

- понятие глобального состояния системы, модифицируемое командами (а не утилитами);

- пользовательский интерфейс, основанный на понятиях текста и команды;

- повышение надежности, благодаря использованию надежного языка программирования и запрещению явного возвращения памяти.

В то же время система Оберон обладает рядом существенных недостатков, самый важный из них – это ограничение расширяемости системы. Фактически, система Оберон поддерживает только одну возможность расширения – расширение окна. Все остальные точки расширения вводятся в приложениях. Основная часть системы является статической и не может быть расширена и адаптирована.

Основной причиной такого ограничения расширяемости является проблема эффективности (см. [34]). Использование сообщений (расширяемых записей) для реализации расширяемых объектов достаточно дорого по времени исполнения. Переход к языку Оберон-2 и использование методов (type-bound procedure) позволил нам сделать расширяемыми практически все объекты системы.

Перечислим другие недостатки системы Оберон:

– механизм финализации реализован в ограниченном объеме, и не может быть использован в расширениях;

– система не поддерживает постоянные объекты, приводя к тому, что различные приложения вынуждены определять такие объекты;

– слабый уровень графики, неадекватный графический интерфейс;

– реализация только неперекрывающихся окон (tiling windows), что усложняет (вернее делает невозможным) использование системы на мониторах с малым разрешением;

– проблемы с переносимостью, ввиду отсутствия четкого разделения между системно-зависимыми и системно независимыми компонентами.

Все эти недостатки существенно сказываются на распространении системы. При проектировании системы Мифрил мы пытались преодолеть недостатки, сохраняя достоинства системы.

Основные отличия системы Мифрил:

- поддержка расширяемости на всех уровнях системы;
- динамическое создание объекта по типу;
- универсальный механизм финализации;
- унифицированный механизм сохранения контекста ошибки;
- поддержка постоянных объектов;
- универсальный расширяемый механизм ввода/вывода;
- поддержка базового набора графических примитивов;
- оконная система, поддерживающая иерархические перекрывающиеся окна.

Разработка базового набора классов системы Мифрил в настоящее время завершена. В следующих версиях системы мы планируем добавить понятия процесса и возможности (квази-)параллельного исполнения.

Заключение

В работе были рассмотрены вопросы построения расширяемых и переносимых инструментальных систем. Работа включала в себя следующие подзадачи:

– анализ пригодности языков программирования на соответствие требованиям расширяемых систем и требованиям надежности;

– выбор схемы трансляции и реализация семейства переносимых компиляторов;

– анализ существующих расширяемых систем, выявление недостатков и способов их преодоления;

– реализация ядра РПС;

– отработка методов переноса компиляторов и системы.

Перечислим основные результаты исследований и реализации:

1) Выработан набор критериев, которым должен удовлетворять язык реализации РПС. Показано, что язык Оберон-2 удовлетворяет этим критериям.

2) Предложена методика реализации мало-языковых транслирующих систем, заключающаяся в фиксации языкового надмножества набора семантически близких языков, построении анализатора для этих языков в единое внутреннее представление и реализации генерации с внутреннего представления для каждой платформы, на которую необходимо перенести компиляторы. Показано, что для пары близких языков (в нашем случае Модула-2 и Оберон-2) трудозатраты на реализацию семейства переносимых компиляторов ненамного превышают затраты на разработку

переносимого компилятора для одного языка. Выполнен анализ схем трансляции и обоснован выбор такой схемы, при которой внутреннее представление между фазами анализа и синтеза является синтаксическим деревом в памяти.

3) Проведена разработка внутреннего представления и реализованы анализаторы языков Модула-2 и Оберон-2. Эти языки достаточно близки как синтаксически, так и семантически, что позволило выделить общую часть, включающую такие компоненты, как лексический анализ, обработка ошибок, таблицу символов, импорт/экспорт.

4) Проведена разработка генерации кода для процессоров i386/486 и Кронос, а также генерация текста на языке ANSI C, что позволяет переносить транслирующую систему на любую платформу, для которой есть ANSI C компилятор (то есть на любую платформу, кроме ЭВМ Кронос). Простота реализации такой генерации в первую очередь обусловлена выбором синтаксического дерева в качестве внутреннего представления.

5) Проведен анализ расширяемых систем на примере системы Оберон. Выявлен ряд недостатков системы Оберон (см. 4.7). Показана важность реализации в ядре системы механизмов финализации и постоянных объектов.

6) Выполнено проектирование ядра РПС. Показано, как с использованием только одиночного наследования можно построить универсальные расширяемые механизмы ввода/вывода, сохранения контекста ошибок, и т.д. Основное внимание уделено поддержке разработки прикладных систем. Ядро РПС определяет и реализует набор механизмов, позволяющих существенно упростить разработку приложений.

7) Совместно с А. Никитиным (оконная система, графика) и А. Хапугиным (динамическая поддержка, загрузчик) реализовано ядро РПС Мифрил.

8) Выполнен перенос системы Мифрил в среду MS-DOS.

Достаточно интересными представляются возможности развития системы. В первую очередь, это реализация генераторов кода для распространенных RISC процессоров и перенос системы Мифрил в развитые обстановки, такие как X-Windows, OS/2 и Windows NT.

Л И Т Е Р А Т У Р А

1. Grady Booch, Object-Oriented Design with Application, Don Mills, ON: Bejamin/Cummings Publishing, 1991
2. Bertrand Meyer, Object-Oriented Software Construction, Englewood Cliffs, NJ: Prentice-Hall International, 1988.
3. Goldberg A. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley, Reading, MS, 1983.
4. Stroustrup B. The C++ Programming Language. Addison-Wesley, 1987. - 328 p.
5. Norman Young, Two Models of Object-Oriented Programming and The Common Lisp Object System, ACM Sigplan Notice, v27, number 4, 1992, p27-36.
6. Szyperski C. The Carrier/Rider Separation. A New Structuring Concept for Open Operating Systems. Geneva, 1991, 16 p. ECOOP'91.
7. Вирт Н. Программирование на языке Модула-2: Пер. с англ. - М.: Мир, 1987. - 222 с.
8. Nelson G. The Programming Language Modula-3 (revised report). CA, 1990 - 83 p., - (Prepr. / DEC SRC. N 39).
9. Reiser M., Wirth N. Programming in Oberon. Steps beyond Pascal and Modula. Addison-Wesley, ACM Press, 1992. - 320 p.
10. Moessenboeck H., Wirth N. The programming language Oberon-2. Structured Programming, 1991 - 179.
11. Rovner P. Extending Modula-2 to Build Large, Integrated System. IEEE Software 3(6), Nov. 1986.
12. Методика разработки многоязыковых трансляторов на примере системы БЕТА. / Ершов А.П., Касьянов В.Н., Покровский С.Б., Поттосин И.В., Степанов Г.Г. – В сб.: Математическая теория

и практика систем программного обеспечения / труды сов.-болг. совещания, Новосибирск, 1982, с. 64-81.

13. Creiler R. OP2: A Portable Oberon Compiler. Zurich, 1990, - 57 p. - (Prepr. / ETH Zurich, Dept. Informatik; N 125).
14. Templ J. SPARC-Oberon. User's Guide and Implementation. Zurich, 1990, 28 p. (Prepr. / ETH Zurich, Dept. Informatik; N 133).
15. Franz M. MacOberon Reference Manual. Zurich, 1990, 32 p. - (Prepr. / ETH Zurich, Dept. Informatik; N 142).
16. ОС Excelsior. Руководство программиста.
17. Кузнецов Д.Н., Недоря А.Е., Тарасов Е.В., Филиппов В.Э. КРОНОС – автоматизированное рабочее место профессионального программиста. В сб.: Автоматизированное рабочее место программиста. Новосибирск, 1988, с. 49-67.
18. Кузнецов Д.Н., Недоря А.Е., Тарасов Е.В., Филиппов В.Э. КРОНОС: семейство процессоров для языков высокого уровня. Микропроцессорные средства и системы, 1989, N 6.
19. Hartmann A.C., A Concurrent Pascal Compiler for minicomputers, Lecture Notice in CS, Springer-Verlag, 1977.
20. J. Gutknecht. Compilation of Data Structures: A New Approach to Efficient Modula-2 Symbol Files. Zurich, 1985, 37 p. - (Prepr. / ETH Zurich, Dept. Informatik; N 64).
21. Кузнецов Д.Н., Недоря А.Е. Проектирование таблиц символов для языков со сложными правилами видимости. В сб.: Методы трансляции и конструирования программ. ВЦ СОАН СССР, Новосибирск, 1984, с. 153-158.
22. Кузнецов Д.Н., Недоря А.Е. Симфайлы как интерфейс операционной системы. В сб.: Информатика. Технологические аспекты. ВЦ СОАН СССР, Новосибирск, 1987, с. 68-75.

23. Pfister C. (ed) Oberon Technical Notes. - Zurich, 1991. 53 p. - (Prepr. / ETH Zurich, Dept. Informatik; N 157).
24. Денисов А.С., Шатохин О.Н. Оберон-2 компилятор для старших моделей IBM PC. В сб.: Среда программирования: методы и инструменты. ИСИ СО РАН, Новосибирск, 1992, с. 114-118.
25. Gutknecht J. The Oberon Guide: System Release 1.2. Zurich, 1990. - 60 p. - (Prepr. / ETH Zurich, Dept. Informatik; N 138).
26. Szyperski C. Write: An Extensible Text Editor for the Oberon System. Zurich, 1991, 47 p. - (Prepr. / ETH Zurich, Dept. Informatik; N 151).
27. Heeb B. Design of the Processor-Board for Ceres-2 Workstation, Report 93, ETH Zurich, 1988.
28. Heeb B., Pfister C. Chameleon: A Workstation of a Different Colour. В печати.
29. Brandis M., Grieler R., Franz M., Templ J. The Oberon System Family. В печати.
30. H. Mossenbock, She: A Simple Hypertext Editor for Programs. Report 145, ETH Zurich, 1990.
31. C. Pfister, The Graphics Editor Condor and The Layout System Pedro. Report 124, ETH Zurich, 1990.
32. Недоря А., Никитин А. Mithril - переносимая Оберон-2 система. В сб.: Среда прогрммирования: методы и инструментыю ИСИ СО РАН, Новосибирск, 1992, с. 99-109.
33. J.R.R. Tolkien. The Lord of the Rings.
34. Szypersky C. Insight ETHOS: On Object-Orientation in Opereting Systems. Diss. ETH Nr. 9884.
35. B. Heeb, C. Pfister. An Integrated Heap Allocator/Garbage Collector. In report 157, ETH Zurich, 1991.

36. Templ J. A Symmetric Solution to the Load/Store Problem. In report 157, ETH Zurich, 1991.

37. Никитин А., Недоря А. Оконная поддержка в системе Mithril. В сб.: Среда прогрммирования: методы и инструменты. ИСИ СО РАН, Новосибирск, 1992, с. 119-125.

38. Marais J. The GADGETS User Interface Management System. Zurich, 1990, 28 p. - (Prepr. / ETH Zurich, Dept. Informatik; N 144).

Приложение А

Модели ООП и языки CLOS и Оберон-2

Таблица отношения моделей ООП G. Bosch [1] и B. Meyer [2] и языков программирования. Таблица взята из [5] и дополнена столбцом для языка Оберон-2.

Features: Essential Desirable Permissible Inadmissible

	BOOP	MOOP	CLOS	O2
routines				
as free subprograms	D	P	Y	Y
exclusively as methods	P	D	N	N
objects				
using relationships	E	E	Y	Y
containing relationships	E	E	Y	Y
reference semantics	P	D	Y	Y
value semantics	P	D	Y	Y
classes				
for abstraction	E	E	Y	Y
with encapsulation	E	E	Y	Y
as types	P	E	Y	Y
as modules	n/a	E	N*	N*
shared variables	P	D	Y	Y
metaclasses	P	P	Y	N
documentation	P	D	Y	Y
assertions	P	D	N	N
generic	P	D	N	N
inheritance				
single	E	E	Y	Y
multiple	P	E	Y	N*
repeated	P	E	Y	N*
redefinition	D	E	Y	Y
typing				
strong	D	E	N*	Y
weak	P	I	Y	N
static	D	D	option	Y
dynamic	D	E	Y	Y
polymorphism				
overloading	D	E	Y	N
parametric	D	E	Y	Y
exceptions				
definable	P	D	Y	N
memory management				
garbage collection	P	E	Y	Y
persistence				
time-persistent objects	D	P	indir.	indirectly
space-persistent objects	D	P	N	indirectly
concurrency				
concurrent objects	D	P	N	N

Пояснения к терминам:

routines
  as free subprograms
        subprograms may be defined independently of any class
  exclusively as methods
        all subprograms are defined as methods within a class

---

objects
  using relationships
        objects may make use of the service offered by other objects
  containing relationships
        object may contain other objects as part of their structure
  reference semantics
        it is possible to define and manipulate references to objects
  value semantics
        it is possible to refer to the value of objects, e.g. assignment, comparison

---

classes
  for abstraction
        class interfaces are defined independently to their implementation
  with encapsulation
        authentication and secrecy are enforced
  as types
        all classes have an associated type
  as modules
        classes comprises a basic unit in softwares physical structure
  shared variables
        it is possible to share a single variable among objects in a class
  metaclasses
        classes may be defined as instance of metaclasses
  documentation
        facilities exist to document classes and to extract this documentation
  assertions
        class semantics are defined by a set of axiomatic assertions
  generic
        class definitions may be parameterized by other class or subprograms

---

inheritance
  single
        classes may inherit the structure or behaviour of one other class

```
  multiple
        classes may inherit the structure or behaviour of more
        than one other class
  repeated
        it is possible to inherit from the same class in more
        than one way at once
  redefinition
        classes may redefine the implementation of inherited
        structure or behaviour
-----------------------------------------------------------------
typing
  strong
        expressions are necessarily type-consistent
  weak
        expressions are not necessarily type-consistent
  static
        names are bound to types during compilation
  dynamic
        names are bound to types during execution
-----------------------------------------------------------------
polymorphism
  overloading
        subprograms are distinguished by their argument and
        return types
  parametric
        methods may respond differently according to an object's
        class
-----------------------------------------------------------------
exceptions
  definable
        it is possible to define program behaviour to failure
-----------------------------------------------------------------
memory management
  garbage collection
        memory is automatically recovered after objects become
        inaccessible
-----------------------------------------------------------------
persistence
  time-persistent objects
        objects may exist across invocations of manipulating
        programs
  space-persistent objects
        objects may be moved among the address spaces of
        distributed processors
-----------------------------------------------------------------
concurrency
  concurrent objects
        object semantics are preserved within multiple threads
        of execution
```

Приложение В

Аксиоматика "хороших" систем

> И увидел Бог все, что Он создал, и вот, хорошо весьма. И был вечер, и было утро: день шестой.
>
> Быт. 1,31

> И будет там большая дорога, и путь по ней назовется путем святым...
>
> Ис. 35,8

> Кто имеет уши слышать, да слышит!
>
> Мат. 13,9

Полувековой мечтой всего программистского сообщества (да не дрогнет рука автора, выписывающего эти слова) является создание такой среды программирования, которая будет устраивать почти всех и работать почти на всех машинах. Как и любая другая N-вековая мечта (для N>=0.5), эта мечта с трудом поддается осуществлению и любое приближение к реализации ее должно рассматриваться как важный шаг вперед.

Переведем обе составляющие этой мечты на более профессиональный язык. Будем называть систему "хорошей", если она асимптотически осуществляет мечту.

Теорема 1. Любая завершенная система не является "хорошей".

Доказательство, к сожалению, требует слишком много места и выходит за рамки данного научного труда.

Замечание.
Хорошей иллюстрацией к данному утверждению может служить Вселенная, созданная Богом за 6 дней.
Конец замечания.

Следствие 1. Разработка "хорошей! системы должна продолжаться всегда.

Рассмотрим теперь проблему с другой стороны.

Аксиома 1. Все люди разные.
Аксиома 2. Но не совсем.

Замечание.
Эти факты (вероятно) смогут быть доказаны с точки зрения генетики, но в нашем случае спокойней считать их аксиомами.
Конец замечания.

Следствие 2. Так как машины делают люди, то все машины получаются разными.

Следствие 3. Но все-таки у них есть что-то общее.

Следствие 4. ***

Очень секретное замечание (перед прочтением закрыть глаза).
Следствие 4 опущено из соображений морали.
Конец очень секретного замечания (глаза можно открыть).

И, наконец:

Следствие 5. "Хорошая" система должна предоставлять возможности развития и адаптации под различные требования различных пользователей. Реализация "хорошей" системы должна опираться на общие свойства людей и машин и, одновременно, позволять учитывать специфические свойства и тех, и других.

Отдавая дань традициям, будем называть главное свойство переносимой системы "расширяемостью", когда речь идет о людях, и "переносимостью", когда речь идет о различных машинах.

В некотором смысле, любая программная система является расширяемой. Мы же будем называть расширяемой системой только такую систему, в которой при добавлении новых возможностей не возникает необходимость изменения базисных понятий и механизмов системы.

Берегитесь лжепророков, которые приходят к вам в овечьей одежде, а внутри суть волки хищные.
Мат. 7,15