ВВЕДЕНИЕ

Для начала следует разобраться, что такое статический анализ и зачем он нужен. Статический анализ кода — анализ программного обеспечения, производимый (в отличие от динамического анализа) без реального выполнения исследуемых программ. В большинстве случаев анализ производится над какой-либо версией исходного кода, хотя иногда анализу подвергается какой-нибудь вид объектного кода, например P-код или код на MSIL. Термин обычно применяют к анализу, производимому специальным ПО. В зависимости от используемого инструмента глубина анализа может варьироваться от определения поведения отдельных операторов до анализа, включающего весь имеющийся исходный код. Способы использования полученной в ходе анализа информации также различны — от выявления мест, возможно содержащих ошибки, до формальных методов, позволяющих математически доказать какие-либо свойства программы (например, соответствие поведения спецификации). Некоторые люди считают программные метрики и обратное проектирование формами статического анализа. Получение метрик и статический анализ часто совмещаются, особенно при создании встраиваемых систем. В последнее время статический анализ всё больше используется в верификации свойств ПО, используемого в компьютерных системах высокой надёжности, особенно критичных для жизни. Также применяется для поиска кода, потенциально содержащего уязвимости .

Более систематично — анализ кода это возможность программы прочитать код анализируемой программы в какой-либо форме, «понять» его и выдать какую-то информацию. Соответственно, практически все анализаторы кода можно представить себе как поиск в определенном представлении программы (возможно с преобразованиями) определенных паттернов и дальнейший подробный анализ найденных участков.

Статический анализ постоянно применяется в следующих областях:

1) ПО для медицинских устройств.

2) ПО для ядерных станций и систем защиты реактора

3) ПО для авиации (в комбинации с динамическим анализом)

Большинство компиляторов (например, GNU C Compiler) выводят на экран «предупреждения» (warnings) — сообщения о том, что код, будучи синтаксически правильным, скорее всего, содержит ошибку. Например:

```
int x;
int y=x+2; // Переменная x не инициализированна
```

Это простейший статический анализ. У компилятора есть много других немаловажных характеристик — в первую очередь скорость работы и качество машинного кода, поэтому компиляторы проверяют код лишь на

очевидные ошибки. Статические анализаторы предназначены для более детального исследования кода.

Типы ошибок, обнаруживаемых статическими анализаторами:

1) Неопределённое поведение — неинициализированные переменные, обращение к NULL-указателям. О простейших случаях сигнализируют и компиляторы.

2) Нарушение блок-схемы пользования библиотекой. Например, для каждого fopen нужен fclose. И если файловая переменная теряется раньше, чем файл закрывается, анализатор может сообщить об ошибке.

3) Типичные сценарии, приводящие к недокументированному поведению. Стандартная библиотека языка Си известна большим количеством неудачных технических решений. Некоторые функции, например, gets, в принципе небезопасны. sprintf и strcpy безопасны лишь при определённых условиях.

4) Переполнение буфера — когда компьютерная программа записывает данные за пределами выделенного в памяти буфера.

```
void doSomething(const char* x)
{
   char s[40];
   sprintf(s, "[%s]", x);    // sprintf в локальный буфер, возможно переполнение
   ....
}
```

5) Типичные сценарии, мешающие кроссплатформенности.

```
Object *p = getObject();
int pNum = reinterpret_cast<int>(p);    // на x86-32 верно, на x64 часть указателя будет потеряна; нужен size_t
```

6) Ошибки в повторяющемся коде. Многие программы исполняют несколько раз одно и то же с разными аргументами. Обычно повторяющиеся фрагменты не пишут с нуля, а размножают и исправляют.

```
dest.x = src.x + dx;
dest.y = src.y + dx;  // Ошибка, надо dy!
```

7) Ошибки форматных строк — в функциях наподобие printf могут быть ошибки с несоответствием форматной строки реальному типу параметров.

```
std::wstring s;
printf ("s is %s", s);
```

8) Неизменный параметр, передаваемый в функцию — признак изменившихся требований к программе. Когда-то параметр был задействован, но сейчас он уже не нужен. В таком случае программист может вообще избавиться от этого параметра — и от связанной с ним логики.

```
void doSomething(int n, bool flag)   // flag всегда равен true
{
```

```
  if (flag)
  {
     // какая-то логика
  } else
  {
     // код есть, но не задействован
  }
}

doSomething(n, true);
...
doSomething(10, true);
...
doSomething(x.size(), true);
```

9) Прочие ошибки — многие функции из стандартных библиотек не имеют побочного эффекта, и вызов их как процедур не имеет смысла.

```
std::string s;
...
s.empty();     // код ничего не делает; вероятно, вы хотели s.clear()?
```

1 АНАЛИЗ И ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ

Можно ввести «типологию» средств анализа кода на основе того, какое представление кода они анализируют и насколько глубоко они его понимают. К примеру, если взять две наиболее распространенные системы (PMD и Findbugs), то PMD работает с представлением программы в виде AST дерева, FindBugs — в виде байт-кода, JavaChecker работает с представлением программы в виде «семантического дерева», где информация из AST дополненна доступом к семантическому контексту. Соответственно, скажем, написать проверку соглашений о стандарте кодирования на JavaChecker и PMD — тривиально, на FindBugs — невозможно. И наоборот — проверку возможности обращения к нулевой ссылке на PMD написать невозможно, на JavaChecker или FindBugs — довольно просто.

Основное внимание хотелось бы обратить на Coco/R – программное обеспечение генерации компиляторов и интерпритаторов. Программа устроена таким образом, что с помощью расширенного представления формы Бэкуса – Наура можно описать любой язык принадлежащий к LL(k) – грамматике. Описание происходит в специальном FRM файле. Описываются лексемы и правила, на основе которых происходит сканирование, анализ синтаксических ошибок, и парсинг – анализ семантических ошибок. Так же можно описать эмуляцию стековой машины и тем самым сгенерировать компилятор. Анализ происходит в 2 этапа – первичный анализ – сканер, обнаружение синтаксических ошибок. На этом этапе генерируется P-код, промежуточный код для стековой машины. Однако в случае если код определить не возможно – если не известны адреса, или случай LL(k) – конфликта – эти проблемы решаются на втором проходе, где анализируется таблица символов, обнаруживаются семантические ошибки и окончательно происходит генерация P-кода. На каждом из проходов существует генерация отчёта об ошибках. Разберём это немного подробнее.

Для каждого проекта, пользователь должен создать текстовый файл, чтобы описать грамматику. Имеет смысл работать в рамках каталога проекта. Файл должен иметь расширение ATG, например CALC.ATG. Помимо грамматики, Coco/R должен считывать frame файлы, в которых описывается сканер и парсер – файл драйвер. Расширение файла FRM, например COMPILER.FRM. В приложени стоит использовать то же имя, что и название файла грамматики – CALC.FRM. После того, как входные файлы были подготовлены, приложение можно запустить с помощью команды:

COCOR CALC.ATG

В строке можно указывать ряд опций:

COCOR –L –C CALC.ATG

В случае, если грамматика описана без ошибок, то на выходи мы должны получить следующее:

1) Сканер FSA (например CALC.HPP и CALCS.CPP)

2) Рекурсивный парсер (например CALCP.HPP и CALC.CPP)

3) Шаблонный драйвер (например CALC.CPP)

4) Список сообщений об ошибках (например CALCE.H)

5) Файл связи токенов с целыми числами, с помощью которых они могут быть иденцифецированы

6) Анализатор (например CALCC.H)

Далее с помощью этих файлов можно скомпилировать различные части приложения и связывать их вместе.

Допустим необходимо создать простой арифмометр который может добавлять ранжированные в группе числа в подсумму, а затем эти результаты добавить к работающему итогу либо отклонить их.

Вход:

```
clear
 // Старт
10 + 20 + 3 .. 7 accept
 // Один промежуточный итог 10 +20 +3 +4 +5 +6 +7 , принимается
3.4 + 6.875..50  cancel
 // Ещё один - отвергнут
3 + 4 + 6 accept
 // Принимается
total // Вывод общей суммы и завершение
```

Правильный ввод этой формы можно описать с помощью простой LL(1) грамматики в Coco/R следующим образом:

```
COMPILER Calc
CHARACTERS
        digit = "0123456789" .
TOKENS
        number= digit { digit } [ "." digit { digit } ] .
PRODUCTIONS
        Calc = "clear" { Subtotal } "total" .
        Subtotal = Range { "+" Range } ( "accept" | "cancel" ) .
        Range = Amount [ ".." Amount ] .
        Amount = number .
END Calc.
```

В целом грмматика может быть описана по правилама EBNF — расширение формы Бэкуса – Наура:

```
 "COMPILER" GoalIdentifier
        ArbitraryText
        ScannerSpecification
        ParserSpecification
"END" GoalIdentifier "." .
```

Описание сканера:

```
ScannerSpecification =
 {
 CharacterSets
        | Ignorable
        | Comments
        | Tokens
        | Pragmas
        | UserNames
} .
```

Список символов:

```
CharacterSets = "CHARACTERS" { NamedCharSet } .
NamedCharSet = SetIdent "=" CharacterSet "." .
CharacterSet = SimpleSet { ( "+" | "-" ) SimpleSet } .
SimpleSet = SetIdent | string | SingleChar [ ".." SingleChar ] | "ANY" .
SingleChar = "CHR" "(" number ")" .
SetIdent = identifier .
```

Комментарии и игнорируемые символы:

```
Comments = "COMMENTS" "FROM" TokenExpr "TO" TokenExpr
[ "NESTED" ] .
Ignorable = "IGNORE" ( "CASE" | CharacterSet ) .
```

Токены:

```
Tokens = "TOKENS" { Token } .
Token = TokenSymbol [ "=" TokenExpr "." ] .
TokenExpr = TokenTerm { "|" TokenTerm } .
TokenTerm = TokenFactor { TokenFactor } [ "CONTEXT" "(" TokenExpr
")" ] .
TokenFactor = SetIdent | string
        | "(" TokenExpr ")"
        | "[" TokenExpr "]"
        | "{" TokenExpr "}" .
TokenSymbol = TokenIdent | string .
TokenIdent = identifier .
```

Прагмы:

```
Pragmas="PRAGMAS" { Pragma } .
Pragma=Token [ Action ] .
Action="(." arbitraryText ".)" .
```

Пример:

PRAGMAS
page = "page" .(. printf("\f"); .)

Имена пользователей:
UserNames = "NAMES" { UserName } .
UserName = TokenIdent "=" (identifier | string) "." .
Пример:
NAMES
period = "." .
ellipsis = "..." .

Интерфейс сканера представляет собой набор функций, которые могут быть вызваны парсером, всякий раз, когда ему необходимо получить токен. Формат спецификации парсера определяется так же по правилам EBNF:

ParserSpecification = "PRODUCTIONS" { Production } .
Production = NonTerminal [FormalAttributes]
[LocalDeclarations] (* Modula-2 and Pascal *)
"=" Expression "." .
FormalAttributes = "<" arbitraryText ">" | "<." arbitraryText ".>" .
LocalDeclarations = "(." arbitraryText ".)" .
NonTerminal = identifier .

Так же Coco/R поддерживает возможность восстановления, в случае обнаружения ошибки с помощью метода Вирта, предложенным в 1986 году. Восстановление происходит только в случае небольшлго количества точек синхронизации в грамматике.

Coco/R выполняет несколько тестов, чтобы удостоверится, что входной тест соответствует заявленной грамматике.

1) Каждый нетерминал связан с одной производственной;
2) Нет безполезный производсвенных;
3) Грамматика — свободный цикл;
4) Все токены отличаются друг от друга;

Если любой из этих тестов нарушен — генерируется ошибка. Семантические ошибки выдаются вконце компиляции. Интерфейс парсера состоит из набора функций, которые могут быть вызваны из сканера.

2 ПОСТАНОВКА ЗАДАЧИ

2.1 Цель проекта

Целью данного дипломного проекта является разработка программаного средсвта статического анализа кода языка относящегося к LL(1) грамматике. Необходимо полностью развернуть все этапы анализа, отразить всё в соответствующих схемах и приложениях. Для выполнения данной задачи, будет разработан интерпритатор языка LL(1) грамматики, с возможностью использования параллельного программирования, использованием процедур и функций.

Опишем семантику анализируемого языка с помощью EBNF:

```
COMPILER Lang
IGNORE CASE
IGNORE CHR(9) .. CHR(13)
COMMENTS FROM "(*" TO "*)"
CHARACTERS
    cr=CHR(13) .
    lf=CHR(10) .
    letter="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopq
rstuvwxyz" .
    digit="0123456789" .
    instring=ANY - "'" - cr - lf .
TOKENS
    identifier = letter { letter | digit } .
    number= digit { digit } .
    string= "'" (instring | "''") { instring | "''" } "'" .
PRODUCTIONS
    Lang= "PROGRAM" identifier ";" Block "." .
    Block= { ConstDeclarations | VarDeclarations }
        CompoundStatement .
    ConstDeclarations = "CONST" OneConst { OneConst } .
    OneConst= identifier "=" number ";" .
    VarDeclarations= "VAR" OneVar { "," OneVar } ";" .
    OneVar= identifier [ UpperBound ] .
    UpperBound= "[" number "]" .
    CompoundStatement = "BEGIN" Statement { ";" Statement } "END" .
    Statement= [CompoundStatement | Assignment
        | IfStatement
        | WhileStatement
        | ReadStatement
        | WriteStatement ] .
```

```
        Assignment= Variable ":=" Expression .
        Variable= Designator .
        Designator= identifier [ "[" Expression "]" ] .
        IfStatement= "IF" Condition "THEN" Statement .
        WhileStatement= "WHILE" Condition "DO" Statement .
        Condition= Expression RelOp Expression .
        ReadStatement= "READ" "(" Variable { "," Variable } ")" .
        WriteStatement= "WRITE"
            [ "(" WriteElement { "," WriteElement } ")" ] .
        WriteElement= string | Expression .
        Expression= ( "+" Term | "-" Term | Term ) { AddOp Term } .
        Term= Factor { MulOp Factor } .
        Factor= Designator | number | "(" Expression ")" .
        AddOp= "+" | "-" .
        MulOp= "*" | "/" .
        RelOp= "=" | "<>" | "<" | "<=" | ">" | ">=" .
    END Lang.
```

Подробности правил описания EBNF будут раскрыт позже, сейчас важно отметить основные конструкции языка. Самая распространённая практика проиллюстрировать конструкцию языка написав небольшую программу на нём.

```
    PROGRAM Debug;
        CONST
            VotingAge = 18;
        VAR
            Eligible, Voters[100], Age, Total;
    BEGIN
        Total := 0;
        Eligible := 0;
        READ(Age);
        WHILE Age > 0 DO
        BEGIN
            IF Age > VotingAge THEN
            BEGIN
                Voters[Eligible] := Age;
                Eligible := Eligible + 1;
                Total := Total + Voters[Eligible - 1]
            END;
            READ(Age);
        END;
        WRITE(Eligible, ’ voters. Average age = ’, Total / Eligible);
    END.
```

Синтаксис очень похож на Pascal или Модула-2, однако целью данного дипломного проекта является описание всех этапов анализа, для того, чтобы это можно было применить на реальных языках программирования в дальнейшем. Для простоты количество конструкций было минимизированно, чтобы передать основную суть и идею всех этапов анализа.

2.2 Требования к программному средству

К основным требованиям данного дипломного проекта можно отнести простоту и восприимчевость отчёта об ошибках. Отчёт должен выводится либо в указанный файл либо на экран терминала, в зависимости от выбора пользователя. В отчёте должно быть чётко прописано, где в какой строчке возникла ошибка, идентифецировать тип ошибки.

Так же к требования можно отнести переносимость программного обеспечения с платформы на платформу. Иными словами, ПО должно работать как на UNIX подобных системах, так и в MS-DOS и WINDOWS.

Так же программа должна легко устанавливаться и удаляться в системе. С программой должен быть поставлен базовый пакет инструкций и документации.

3 МОДЕЛИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

3.1 Этапы трансляции

Сами по себе трансляторы очень сложные программы, поэтому нельзя рассматривать процесс трансляции в один этап. Существует аналитическая фаза — в которой программа анализирует синтаксическую и семантическую ограничения источника. За этим следует синтетическая фаза — генерируется объектный код целевого языка. Компоненты транслятора можно разделить на 2 части -Front end и Back end, причем Front end не зависит от целевой машины, а Back end очень сильно от неё зависит. Основные компоненты транслятора изображены на рисунке 3.1

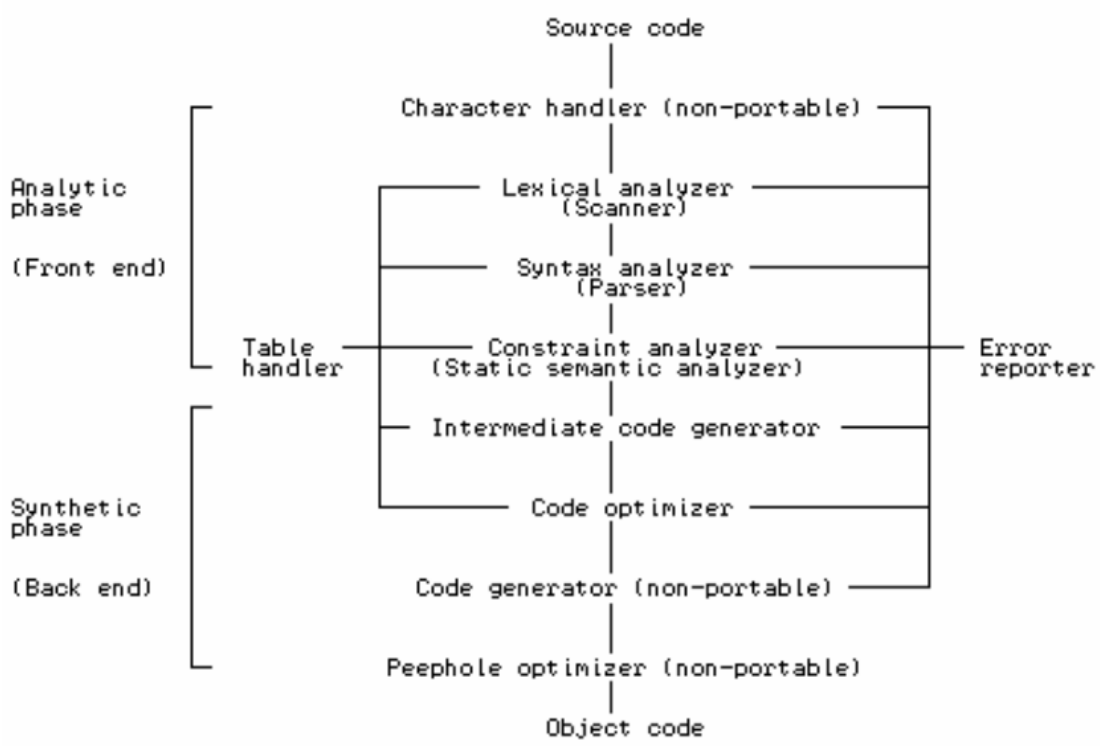

Рисунок 3.1 — Структура и фазы транслятора

Набор символов для обработки варируется от операционной системы. Лексический анализатор или сканер делит исходный текст программы на группы, которые представляют логические лексемы языка, такие как строки, константы, ключевые слова, такие как while if, операторы - <= и т. д. Некоторые из них получаются очень просто на выходе из сканера, некоторые должны быть связаны с различными свойствами, такими как значения. Лексический анализ происходит иногда очень просто, а иногда нет. Пример лексического анализа: WHILE A > 3 * B DO A := A - 1 END

WHILE	keyword	
A	identifier	name A
>	operator	comparison
3	constant literal	value 3
*	operator	multiplication
B	identifier	name B
DO	keyword	
A	identifier	name A
:=	operator	assignment
A	identifier	name A
-	operator	subtraction
1	constant literal	value 1
END	keyword	

Таблица 3.1 — Лексический анализ

Разумно думать, что данное выражение синаксически корректно, однако ни одна программа действительно не имеет смысла до её динамического выполнения. Семантика — это термин, который используется

для описания смысла. Анализатор часто называют статический семантический анализатор или просто — семантический анализатор. Результат синтаксического и семантического анализа часто удобно представлять в виде абстрактного синтаксичесеого дерева — AST. Это очень удобно, так как позволяет оптимизировать некоторые участки на более позднем этапе генерации объектного кода, так же язык может иметь очень много ключевых слов, однако AST показывает только ключевые и значимые компоненты, отражающие смысл программы. AST — дерево лишено семантических подробностей, важно отобразить суть. Семантический анализатор имеет задачу идентифецировать тип и другую контекстную информацию для ращличных узлов. Пример AST дерва:

WHILE (1 < P) AND (P < 9) DO P := P + Q END

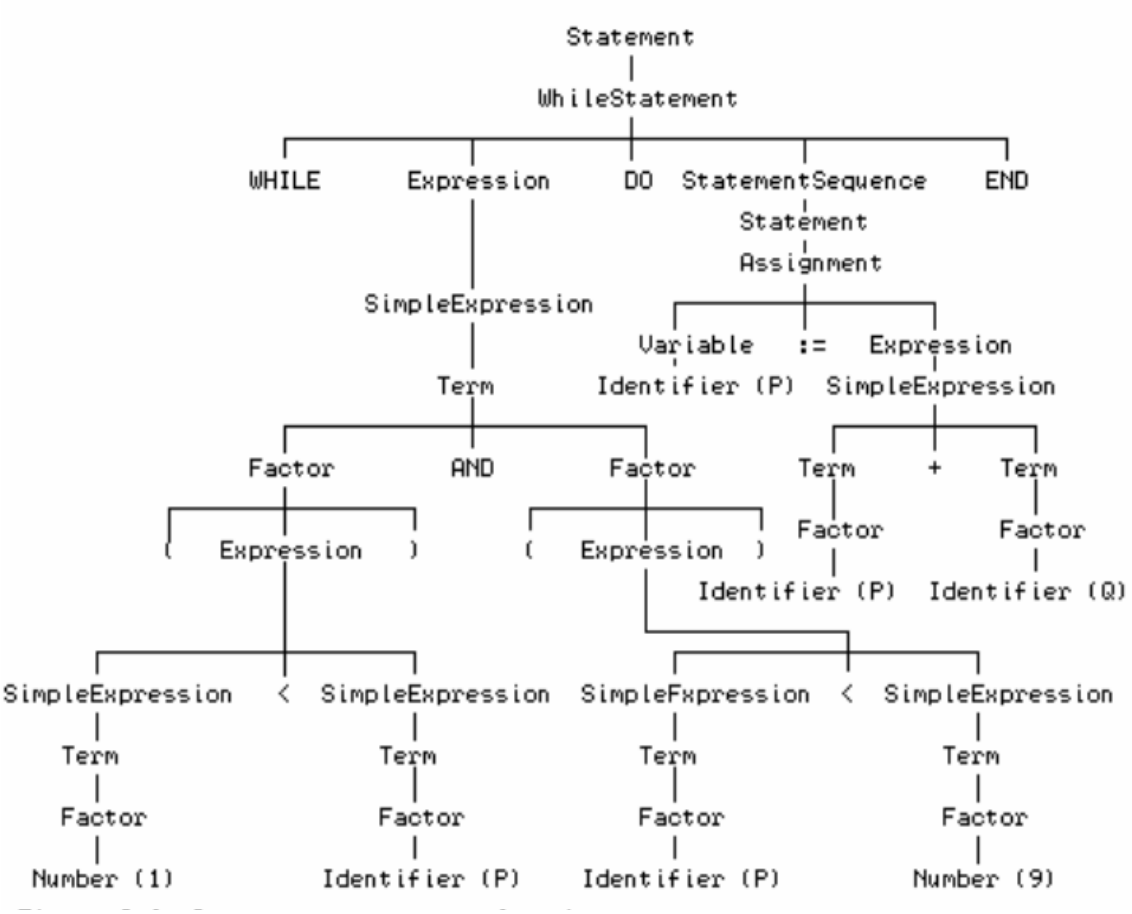

Рисунок 3.2 — AST дерево

Фазы, которые были перечислены выше — носят аналитический характер. Так же есть те, которые носят более синтетический. Первой из них может быть промежуточная генерация кода. В некоторых случаях, эта фаза частично выполняется на более ранних этапах, или вовсе отсутствуют в простых трансляторах. Здесь используется структура, полученная на ранних

этапах для создания формы кода, в виде скилета или набора макросов или асемблер или даже высоко-уровневый код для обработки внешним компилятором. Основное отличие данного промежуточного кода от фактического машинного это то, что промежуточный код не должен описывать подробно такие вещи, как точная машина, например точные адреса и т. д. Например:

```
L0 if 1 < P goto L1
        goto L3
L1 if P < 9 goto L2
        goto L3
L2 P := P + Q
        goto L0
L3      continue
```

или:

```
L0 T1 := 1 < P
        T2 := P < 9
        if T1 and T2 goto L1
        goto L2
L1 P := P + Q
        goto L0
L2      continue
```

Следующим этапом может быть оптимизация кода. Транслятор может быть снабжен этим механизмом в попытке улучшить промежуточный код в интересах производительности. Например найти зоны, которые никогда не используются и удалить их. Однако это не безопасно.

Наиболее важным этапом в Back end — является обязательная генерация объектного кода. Эта фаза принемает входные данные от предыдущих фаз и производит объектный код выделяя память для данных, выбор регистров для расчета промежуточных результатов, индексации и т. д. Этот этап требует отдельного пристального внимания.

Как известно, пользователи склонны допускать множество ошибок. Поэтому важно предусмотреть систему обработки ошибок, особенно на ранних этапах. Обнаружение ошибок во время компиляции в исходном коде не следует путать с обнаружением ошибок во время выполнения объектного кода. Общая структура анализатора представлена на рисунке 3.3

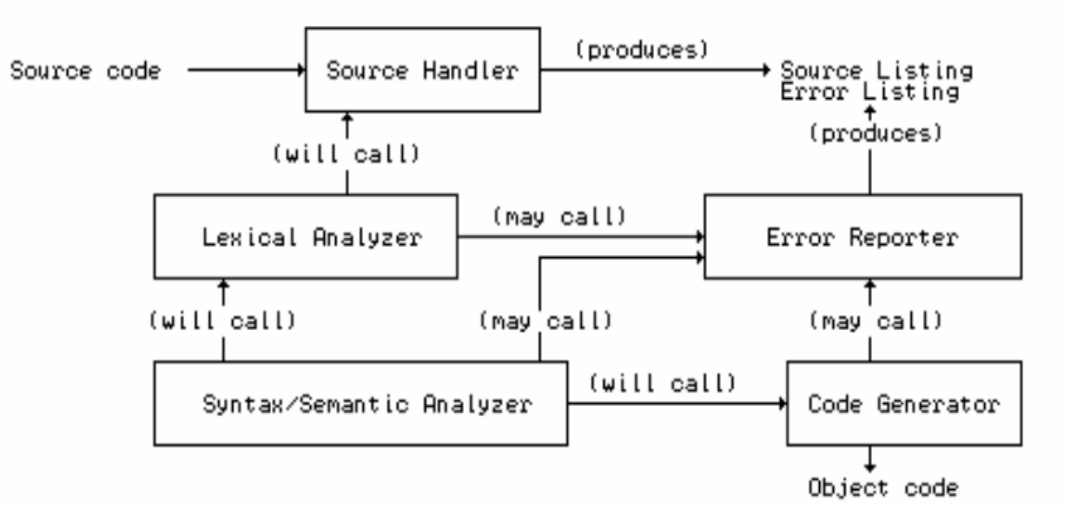

Рисунок 3.3 — Структура анализатора

3.2 Многоступенчатые трансляторы

Несмотря на то, что концептуально транслятор может быть разделён на фазы, некоторые фазы могут быть выполнены одновременно на ранних этапах, т.е. объеденены или чередуются. Традиционно сперва выполняется первоочередной проход по исходному файлу, делаются какие-то преобразования и результат записывается в промежуточный файл, откуда он может быть прочитан при последующем проходе. Эти проходы могут быть обработаны различными частями компилятора. Они могут быть сгенерированы несколькими программами, они могут взаимодействовать используя свои собственные специализированные формы, промежуточный язык, они могут взаимодействовать за счёт внутренних структур данных, а не файлов, они могут делать несколько проходов по исходному коду. Количество проходов зависит от множества факторов. Некоторым языкам необходимо по меньшей мере два прохода, например там, где значение идентификаторов можно определить только после первого прохода. Хотя современные компьютеры наделены достаточным объемом памяти множественный проход может быть важным фактором, если мы проектируем сложный язык в пределах небольшой системы. Многоступенчатые трансляторы могут так же использовать несколько проходов для оптимизации кода, генерации ошибок и обработки ошибок. Тем не менее многоходовые трансляторы как правило медленее однопроходных. На

практике как правило первый проход используют для генерации ассемблер кода или кода более высокого уровня, который потом может быть выполнен используя сторонний компилятор на реальной машине.

3.3 Архитектура простой машины

В этом разделе обсудим гипотетическую эмуляцию машин более подробно. Большинство ЦП(центральные процессоры) чипы, используемые в современных компьютерах, имеют один или несколько регистров и акумуляторов, которые можно рассматривать как локальная память и соответсвтенно могут быть выполнены простые арифметические и логические операции. Эти регистры могут быть однобайтными(8 бит) или более типично для современных компьютеров — небольшая упаковка байт или иметь длинну слова. Одним из основных внутренних регистров является регистр инструкций IR, в который поступают байты инструкций, которые машина должна выполнить. Эти инструкции, как правило чрезвычайно простые, например: «Очистить регистр», «Переместить байт из одного регистра в другой» имеющие типичный порядок сложности. Иснтрукции могут быть определены одним байтом, однако некоторым понадобится дополнительные байты для полного определения. В этих многобайтовых инструкциях, первый обычно обозначает операцию, а остальные относятся либо к значению, над которым нужно произвести действие, или адрес, по которому можно определить значение, над которым нужно выполнить операцию. Простейшие процессоры имеют лишь несколько регистров данных, и очень ограничены в том, что могут на самом деле с этими данными сделать, поэтому процессоры должны взаимодействовать с памятью компьютера, и происходит это с помощью так называемой шины между внутренними регистрами и значительно большего числа внешних устройств памяти. Когда информация должна быть переданы из памяти - процессор помещает соответствующую адресную информацию на адресной шине , а затем передает или принимает сами данные на шину данных . Это проиллюстрировано на рисунке 3.4

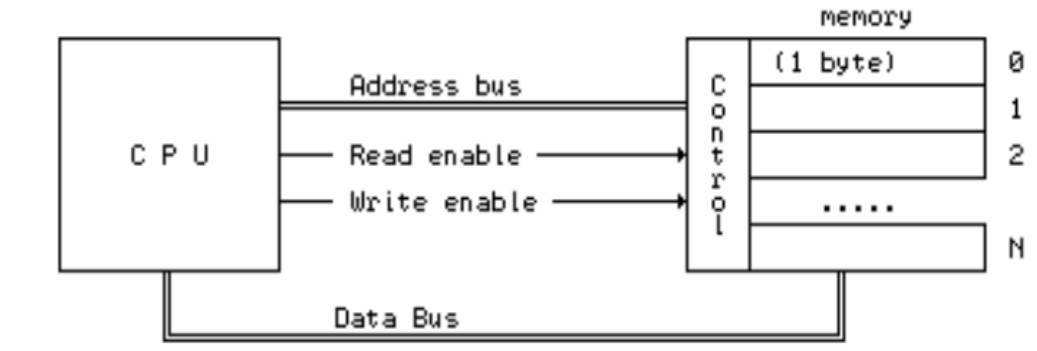

Рисунок 3.4 — Связь CPU с памятью по адресной шине и шине данных

Машина извлекает данные из памяти используя шину данных и помещает её в регистр инструкций, где эта инструкция выполняется.

3.4 Режимы адресации

Из предыдущих разделов можно предположить, что подготовка программы на машинном уровне часто состоит из последовательности простых инструкций, каждая из которых включает операцию машинного уровня и один или несколько параметров. Простой пример на языке высокого уровня может выглядеть так:

AmountDue := Price + Tax;

Некоторые машины и языки ассемблера обеспечивают для таких операций с точки зрения кода трехадресный вид, в котором операция обозначает мнемонику и обычно вызывает опкод операции, включающий два операнда и назначение:

operation destination, operand 1, operand 2

Например:

ADD AmountDue, Price, Tax

Мы также можем выразить это с помощью вызова функции:

destination:=operation(operand 1, operand 2)

Однако во многих машинных реализациях такая возможность отсутствует. На высоком уровне такие операции как Price := Price * InflationFactor; - отражаются на более низком уровне в так называемой двухадресном виде.

Общий вид:

operation destination, operand.

Пример:

MUL Price, InflationFactor

Однако и данный режим адресации отсутствует во многих машинах. Одно из назначений операнда может быть опущено, так как обозначает машинный регистр(другой может обозначать машинный регстр, константу или машинный адрес). Это часто называют полуторным кодом адресации.

Пример:

MOV R1, Value ; CPU.R1 := Value

ADD Answer, R1 ; Answer := Answer + CPU.R1

MOV Result, R2 ; Result := CPU.R2

Наконец, в так называемых аккумуляторных машинах может быть опущен один адрес, где назначением всегда является машинный регистр. В некоторых языках ассемблера такие инструкции могут по-прежнему использовать двухадресную форму, как указано выше.

Пример:

LDA Value ; CPU.A := Value

ADA Answer ; CPU.A := CPU.A + Answer

STB Result ; Result := CPU.B

Хотя для многих из этих примеров может сложиться впечатление, что на машинном уровне операции требуют несколько байт для их представления но это не всегда верно. Например операции, которые включают только машинные регистры:

MOV R1, R2 ; CPU.R1 := CPU.R2

LDA B ; CPU.A := CPU.B

TAX ; CPU.X := CPU.A

Здесь может потребоваться только один байт - как бы наиболее очевидным в языке ассемблера, который раньше

использовался в третьем представлении. Сборка таких программ значительно облегчается простым и самосогласованным обозначением для исходного кода и именно его мы будем рассматривать в дальнейшем.

Инструкции, которые производят манипуляцию над значениями, кроме тех, которые в регистрах машины - как правило являются многобайтовыми. Первый байт обычно определяет саму(и возможно регистр или регистры которые используются) операцию, в то время как остальные байты указывают значения (или адреса памяти других значений), которые участвуют в операции . В таких инструкциях есть несколько способов использования вспомогательного байта. Это разнообразие даёт возможность использовать так называемые различниые режимы адресации для процессора, цель которых обеспечить эффективный адрес для использования в инструкциях. Конечно эти режимы чрезвычайно меняются от процессора к

процессору, и можно упомянуть лишь несколько типичных примеров. Различные возможности можно выделить в некоторых языках ассемблера за счет использования различных мнемоник, которые на первый взгляд кажутся похожими. На других языках ассемблера различие может заключатся в различных синтаксических формах, используемых для определения регистров, адреса или значения . Можно даже найти разные языки ассемблер для одного общего процессора.

Во **внутренней** адресации операнд подразумевается в самом коде операции, и часто инструкция содержится в одном байте. Например, чтобы очистить машинный регистр A:

CLA or CLR A ; CPU.A := 0

Опять же подчеркну, что хотя вторая форма использует две составляющие, это не всегда подразумевает использование двух байтов кода на уровне машины.

В **непосредственной** адресации вспомогательных байт для инструкции обычно определяет адрес по которому должно быть определено значение в регистре.

Пример:

ADI 34 or ADD A, #34 ; CPU.A := CPU.A + 34

В этих двух примерах использование слова «адресс» немного вводит в заблуждение, т.к. значение ничего общего с адресом не имеет. Однако это станет ясно из двух следующих режимов.

В **прямой** или **абсолютной** адресации вспомогательные байт обычно указывает адрес памяти от значения, которое должно быть получено или комбинировано со значением в регистре, или указать, где значение регистра будет храниться. Например:

LDA 34 or MOV A, 34 ; CPU.A := Mem[34]

STA 45 or MOV 45, A ; Mem[45] := CPU.A

ADD 38 or ADD A, 38 ; CPU.A := CPU.A + Mem[38]

Очень часто пользователи путают непосредственную и прямую адресацию, ситуация не улучшилась, так как нет согласованности в нотации между различными языками ассемблера, и даже существует множество способов определять режим адресации. Например, для процессоров 80x86 Intel, которые использовались в IBM-PC и совместимых, низкоуровневый код

записывается в двухадресной форме аналогичный тому, который показан выше, - но непосредственный режим обозначается без необходимости специальных символов как # в то время как режим прямой адресации может иметь адрес в скобках:

ADD AX, 34 ; CPU.AX := CPU.AX + 34 - непосредственная адресация

MOV AX, [34] ; CPU.AX := Mem[34] - прямая адресация

В **косвенной** адресации один из операндов в инструкции определяется как адрес, другой как индексный регистр, значение которого в момент исполнения может рассматриваться как указания индекса в массиве, где хранится этот адрес.

MOV R1, @R2 ; CPU.R1 := Mem[CPU.R2]

MOV AX, [BX] ; CPU.AX := Mem[CPU.BX]

Не все регистры в машине могут быть использованы для данного режима. В самом деле, некоторые машины имеют довольно неудобные ограничения в этом отношении.

Некоторые процессоры позволяют использовать очень мощные вариации на прямой и косвенных способах адресации. Например, в памяти прямой адресации один операнд может указать два адреса памяти, первый из которых дает адрес первого элемента массива, а второй дает адрес переменной, значение которой будет использоваться в качестве индекса в массиве.

MOV R1, 400[100] ; CPU.R1 := Mem[400 + Mem[100]]

Аналогично в памяти косвенной адресации один из операндов в инструкции определяет адрес памяти, по которому будет найдено значение, формирущее эффективный адрес, где должен быть найден другой операнд.

MOV R1, @100 ; CPU.R1 := Mem[Mem[100]]

3.5 Архитектура с 1 - А аккумулятором

Сложные процессоры могут иметь несколько регистров но их основные принципы — эмуляция, поэтому можно проиллюстрировать следующую модель процессора одного аккумулятора.

Схематически мы могли бы представлять эту машину как показано на рисунке 3.5

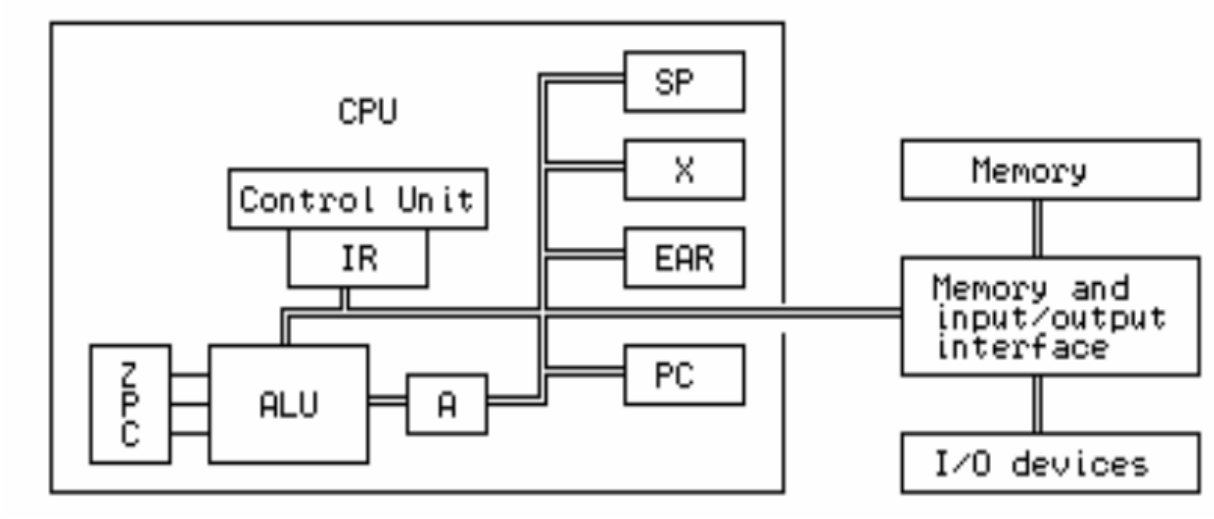

Рисунок 3.5 — Архитектура компьютера с одним аккумулятором

Символы в этой схеме относятся к следующим компонентов машины:

ALU - арифметико-логическое устройство, где арифметические и логические операции на самом деле выполняются

A - 8 - разрядный аккумулятор, регистр для выполнения арифметических или логических операций.

SP - является указатель стека 8-бит, регистр, который указывает на область памяти которая может быть использована в качестве стека.

X - представляет собой 8-битный индексный регистр, который используется в индексации областей памяти, которые концептуально образуют массивы данных.

Z , P, C - одиночные флаги состояний или регистры состояний, которые установлены в значение "истина", когда операция вызывает реестр для изменения к нулевому значению, или в положительное значение, или используется перенос, соответственно.

IR - является 8- разрядным командым регистром, в котором хранится значение байта инструкции которая на данный момент выполняется.

PC - является 8-разрядным счетчиком команд, который содержит адрес в памяти инструкции, которая должна выполняться следующей.

EAR - является эффективным адресным регистром, который содержит адрес байта данных над которыми происходят манипуляции текущей командой.

3.6 Эмуляция для стековой машины

Действия процессора могут быть легко смоделированы на программном уровне. По существу нам нужно только написать эмулятор, модели выборки инструкций, представляющий из себя цикл машины, и мы можем сделать это в любом подходящем языке, для которого у нас уже есть компилятор на реальной машине.

Такие языки, как Modula-2 или C ++, очень подходят для этой цели. Мало того, что у них есть возможности для выполнения побитовых операций, ещё одним важнейшим приемуществом является то, что можно реализовать различные фазы трансляторов и эмуляторов как когерентных, четко разделенных модулей (в Модула-2) или классов (в C++). Расширенные версии Pascal, такие как Turbo Pascal, также обеспечивают поддержку таких модулей в виде блоков. C также очень подходит на первых этапах, но он не очень подходит для реализации определения отдельных модулей, так как механизм файл заголовка используемый в C менее проницаемый, чем механизмы на других языках. При моделировании нашей гипотетической машины в C++ удобно определять интрефейс с помощью открытого интерфейса к классу. Основная задача интерфейса - объявить процедуру эмулятора для интерпретации кода, хранящегося в памяти машины.

Для данного дипломного проекта мы будем рассматривать компилятор, который генерирует объектный код для гипотетической "стековой машины", который может не иметь никаких общих регистров данных такого рода как было описано ранее, но использовать такие функции как, в первую очередь, манипуляции с указателем стека и связанный стек. Данная архитектура идеально подходит для анализа сложных арифметических или логических выражений, а также для осуществления языков высокого уровня, которые поддерживают рекурсию. Для начала необходимо определить архитектуру.

По сравнению с архитектурой машин основаных на базе регистров, эта архитектура может показаться на первый взгляд немного странной, из-за малочисленности регистров. Как и большинство машин мы будем предполагать, что данные храняться в памяти, которую можно смоделировать в виде линейной матрицы. Элементами памяти являются "слова", каждое из которых может хранить одно целое число - как правило, с использованием 16 битного представления и двоичным дополнительным кодом.

Схематически мы могли бы представлять эту машину как показано на рисунке 3.6

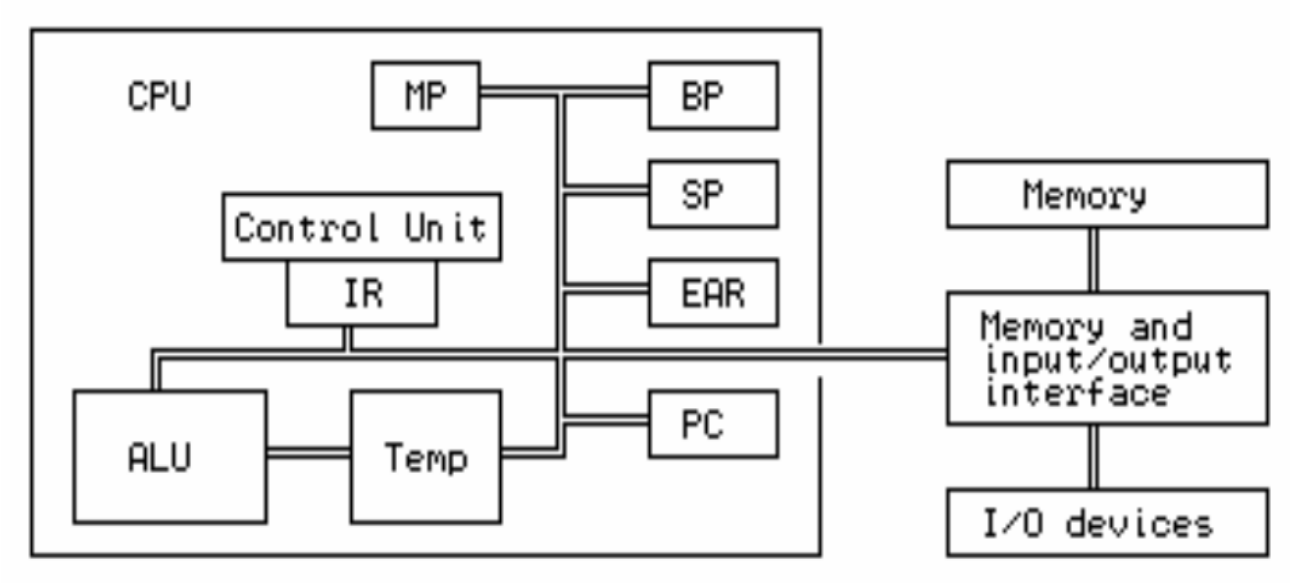

Рисунок 3.6 - Простой стек-ориентированный процессор и компьютер

Символы в этой схеме относятся к следующим компонентов машины:

ALU - арифметико-логическое устройство, где арифметические и логические операции на самом деле выполняются

Temp - представляет собой набор 16 -битных регистров для хранения промежуточных результатов, необходимых во время арифметических или логические операций. Эти регистры не могут быть доступны в явном виде.

SP - 16-битный указатель стека, регистр , который указывает на область памяти используемую в качестве основного стека.

BP - 16-битный указатель базы, регистр, указывающий на базе области памяти в стеке, известный как кадр стека, который используется для хранения переменных.

MP - 16- разрядный указатель стека знак, регистр используется при обработке вызовов процедур. Использование станет очевидным далее.

IR - 16- разрядная командный регистр, в котором хранится адрес инструкции выполняемая в настоящее время.

PC - 16-разрядный счетчик команд, который содержит в памяти адрес инструкция, которая будет выполнена следующей.

EAR - эффективный адресный регистр , который содержит в памяти адрес данных над которыми происходят манипуляции текущей командой.

Для простоты будем считать, что код хранится в нижнем конце памяти, и что верхняя часть памяти используется в качестве стека для хранения

данных. Будем считать, что самый верхний раздел этого стека является символьный пулл, в котором хранятся константы, такие как текстовые строки символов. Ниже этого пулла расположен кадр стека, в котором хранятся статические переменные. Остальная часть стека должена быть использован для оперативной памяти. Типичное распределение памяти отображено на рисунке 3.7, где метки CodeTop и StkTop будут использоваться для обеспечения защиты памяти в эмулируемой системе.

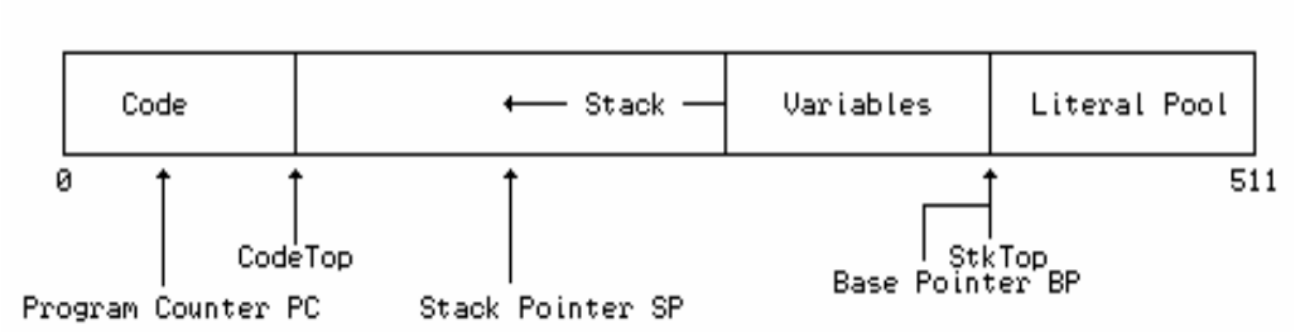

Рисунок 3.7 — Использование памяти в простых стек ориентированных машинах

Мы предполагаем, что программа загрузки загрузит код в нижней части памяти (обозначенный CodeTop), также будут загружены литералы в симпольный пулл (обозначенный StkTop). Далее произойдёт инициализация SP указателя стека и указателя базы BP. Первая инструкция в любой программе будет нести ответственность за резервирование дальнейшего пространство в стеке для своих переменных, просто путем уменьшения указателя стека SP на количество слов, необходимых для этих переменных. Переменную можно определить путем включения смещения на базу ругистра BP. Так как стек "растет вниз" в памяти, от высоких адресов в сторону низких, эти смещения, как правило, представляют собой отрицательные значения.

Минимальный набор комманд для данной машины описан в таблице 3.2. Для удобства обозначения TOS(«Top on stack» - вершина стека), SOS(«Second on stack» - второй в стеке), SP — регистр стека, POP — извлечь из конца стека, PUSH — поместить вконец стека. Некоторые из этих операций принадлежат к категории, известной как нулевой адрес инструкции.

ADD	POP TOS and SOS, add SOS to TOS, PUSH sum to form new TOS
SUB	Выбрать TOS and SOS, subtract TOS from SOS, PUSH result to form new TOS
MUL	POP TOS and SOS, multiply SOS by TOS, PUSH result to form new TOS

DVD	POP TOS and SOS, divide SOS by TOS, PUSH result to form new TOS
EQL	POP TOS and SOS, PUSH 1 to form new TOS if SOS = TOS, 0 otherwise
NEQ	POP TOS and SOS, PUSH 1 to form new TOS if SOS # TOS, 0 otherwise
GTR	POP TOS and SOS, PUSH 1 to form new TOS if SOS > TOS, 0 otherwise
LSS	POP TOS and SOS, PUSH 1 to form new TOS if SOS < TOS, 0 otherwise
LEQ	POP TOS and SOS, PUSH 1 to form new TOS if SOS <= TOS, 0 otherwise
GEQ	POP TOS and SOS, PUSH 1 to form new TOS if SOS >= TOS, 0 otherwise
NEG	Negate TOS
STK	Dump stack to output (useful for debugging)
PRN	POP TOS and write it to the output as an integer value
PRS A	Write the nul-terminated string that was stacked in the literal pool from Mem[A]
NLN	Write a newline (carriage-return-line-feed) sequence
INN	Read integer value, pop TOS, store the value that was read in Mem[TOS]
DSP A	Decrement value of stack pointer SP by A
LIT A	PUSH the integer value A onto the stack to form new TOS
ADR A	PUSH the value BP + A onto the stack to form new TOS. (This value is conceptually the address of a variable stored at an offset A within the stack frame pointed to by the base register BP.)
IND	POP TOS to yield Size; POP TOS and SOS; if 0 <= TOS < Size then subtract TOS from SOS, PUSH result to form new TOS
VAL	POP TOS, and PUSH the value of Mem[TOS] to form new TOS (an operation we shall call dereferencing)
STO	POP TOS and SOS; store TOS in Mem[SOS]
HLT	Halt
BRN A	Unconditional branch to instruction A
BZE A	POP TOS, and branch to instruction A if TOS is zero
NOP	No operation

Таблица 3.2 — Список комманд

Еще раз, для эмуляции такой машины с помощью программы, написанной на C++, будет удобно описать интерфейс к машине с помощью определения класса. Как и в случае с аккумуляторной машиной, основной функцией объекта является выполнение самой эмуляции, но для целесообразности мы будем экспортировать дополнительные объекты,

которые позволяют развивать дальше ассемблер, компилятор и загрузчик, который оставит псевдо-код непосредственно в памяти после трансляции исходного кода. Сам эмулятор моделирует типичную выборку комманд из стека, выполняя цикл фактической машины. Полную реализацию, можно найти в приложении. Структуру класса так же можно посмотреть в диаграмме классов в приложении.

3.7 Генерация ошибок и двойной проход

Как уже было отмечено ранее, анализ кода проходит в несколько этапов, т. к. в некоторых случаях для генерации объектного кода необходимо знать адреса меток, или переменных. Например:

BCC EVEN

код не может быть определён сразу, пока не будет известен адресс EVEN:

EVEN BNZ LOOP

Конечно есть и другой способ — в один проход. В данном способе используется такое понятие, как опережающие ссылки. Однако при такой реализации структура объектного кода может выглядеть очень большой и запутанной.

На первом этапе происходит генерация так называемого опкода — кода операции. Результат прохода записывается в таблицу символов фиксированной длинны. Структура таблицы может быть реализованна в виде динамического стека или хэш таблицы. Удобно ввести понятие строки кода. Пример таблицы для макро ассемблера изображен на рисунке 3.8

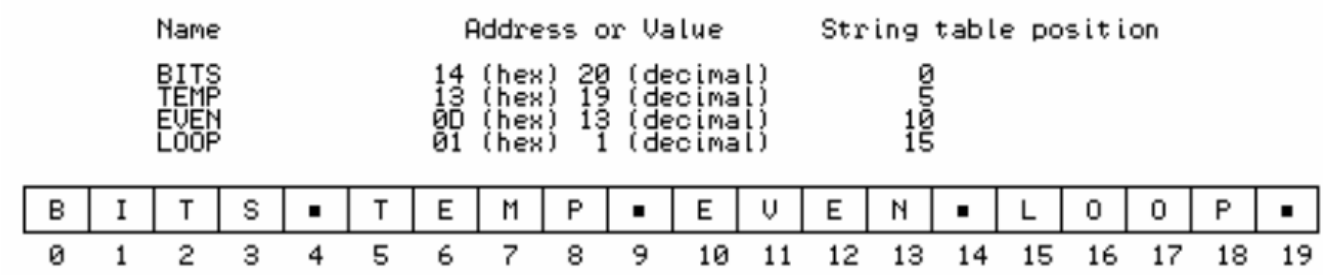

Рисунок 3.8 — Таблица сиволов для простой программы на ассемблер

Много времени может быть потрачено на поиск идентификаторов и мнемоник в таблицах, и целесообразно рассмотреть, как можно улучшить простой линейный поиск. Популярный способ реализации очень эффективный - с помощью хэш-функций. Например:

hash = (ident[first] * ident[last]) % tablemax;

В случае возникновения коллизии — необходимо произвести процедуру коррекции и выполнения функции хэширования заново.

Более подробно о структуре таблицы символов будет описано в разделе проектирования программного средства.

Ошибки могут быть классифицированы на основе стадии, на которой они могут быть обнаружены. На каждом из этапов работает генерация отчёта об ошибках, в случае её обнаружения. В частности, некоторые важные потенциальные ошибки:

1. Ошибки которые могут быть обнаружены исходным обработчиком:

- Преждевременный конец исходного файла - это может быть довольно фатальная ошибка

2. Ошибки, которые могут быть обнаружены с помощью лексического анализатора:

- Используется неузнаваемый символ
- Слишком широкий комментарий
- Название переменной слишком длинное
- Переполнение в формировании числовых констант
- Использование не цифровых символов в числовых литералах
- Первый символ названия переменной не является буквой

3. Ошибки, которые могут быть обнаружены с помощью синтаксического анализатора:

- Использование полностью неузнаваемым символов или неуместными символов, таких как номера, где должны быть комментарии
- Несоблюдение синтаксиса операторов, опуская необходимые параметры в конструкции, и так далее.

4. Ошибки, которые могут быть обнаружены с помощью семантического анализатора:

- Использование неопределенных мнемоник

- Неорпеделенные переменные
- Неорпеделено правильное количество фактических параметров
- Дублирование в определении и так далее.

3.8 Спецификация языка

Изучение синтаксиса и семантики языков программирования могут быть сделаны на многих уровнях, и это является важной частью современной вычислительной техники. Люди используют языки, чтобы общаться. В обычной речи они используют естественные языки, как английский или французский язык; для более специализированных приложений, используются технические языки, например в математике:

$$\forall\, x\, \exists\, \varepsilon :: | x - \xi | < \varepsilon$$

Полезные языки программирования должны удовлетворять описанию и реализации, однако трудно найти языки, которые удовлетворяют обоим требованиям. Языки низкого уровня более эффективны, в то время как языки высокого уровня более восприимчивы для понимания. В последние годы много усилий было потрачено на формализации языков программирования, и в поиске путей для их описания и определения. Конечно, формальный язык программирования должен быть описан с помощью другого языка. Этот язык описания называется - **метаязык**. Ранние языки программирования были описаны с использованием английского языка как метаязыка. Точная спецификация требует, чтобы метаязык был полностью однозначным, и это не сильная черта английского языка.

Натуральные языки, технические языки и языки программирования похожи по нескольким принципам. В каждом случае **предложения** состоят из **последовательности символов** или **токенов** или **слов**, и построение этих предложений регулируется путем применения двух наборов правил:

1. **Синтаксические правила** - описывают форму предложений в языке. Например, в английском языке, предложение "They can fish" синтаксически правильно, в то время как предложение "Can fish they" не соответствует действительности. Возьмем другой пример, язык двоичных цифр используются только символы 0 и 1, расположенных в строках, образованных путём объединения: 101 - синтаксически правильно, в то время как предложение 1110211 - синтаксически некорректно.

2. **Семантические правила** - определяют значение синтаксически правильных предложений в языке. Сам по себе код 101 не имеет смысла без добавления семантических правил о том, что это должно быть

интерпретировано как представление некоторого числа. Предложение "They can fish" более интересено, ибо оно может иметь два возможных значения; набор семантических правил будет еще труднее сформулировать.

Для определения языков программирования следует упомянуть о некоторых особенностях теории формальных языков. Начнем с нескольких абстрактных определений:

1. **Символ** или **токен** — есть атомарная сущность, представленная в виде букв, символов, знаков или иногда с помощью зарезервированных ключевых слов. Например + , ; END.

2. **Алфавит A** — не пустой, но конечный набор символов. Например:

- / * a b c A B C BEGIN CASE END

или

- / * a b c A B C { switch }

3. **Фраза**, **слово** или **строка** над алфавитом A — это последовательность σ=a1a2...an символов из A.

4. Часто бывает полезно предположить существование строки нулевой длины, называемой **пустой строкой** или **пустое слово** (бозначают ε). Она имеет свойство, что, если она расположена слева или справа от любого слова, то слово остается неизменным. aε=εa=a

5. Множество всех слов длины n над алфавитом A обозначается A^n. Множество всех строк над алфавитом A называется **замыканием Клини** или просто **замыканием**, и обозначается A^*. Множество всех строк длиной по крайней мере один символ над алфавитом A называется его **положительным замыканием**, и обозначается A^+. Таким образом

A^* = A^0 U A^1 U A^2 U A^3 …

6. **Язык L** над алфавитом A это подмножество A^*. На современном уровне обсуждения это не имеет никакого смысла. Язык это просто набор строк. Язык, состоящий из конечного числа строк можно определить просто путем перечисления всех этих строк, или предоставление правила для их вывода. Это может быть даже возможным для простых бесконечных языков. Например L = { ([a+)^n (b])^n | n > 0 } - это определяет такие выражения, как:

[a + b]

[a + [a + b] b]

[a + [a + [a + b] b] b]

Некоторые простые языки можно представить в виде **регулярного выражения.** Регулярное выражение представляет собой форму записи, использующую символы из алфавита A в сочетании с некоторыми другими метасимволами, которые представляют собой операции. Однако на практике, конечно большинство языков гораздо сложнее, чем можно представить с помощью регулярного выражения. В частности, регулярные выражения не являются достаточно мощными, чтобы описать языки, которые используют собственные вложения в своих описаниях. Самовложение может быть например в описании структурных операторов, которые включают в себя компоненты, которые так же могут быть операторами, или в выражении, состоящее из факторов, которые могут содержать дополнительные выражения в скобках, или переменные, которые объявлены в терминах некоторого типа, который структурирован на базе других типов и т. д.

Таким образом, мы подошли к понятию **грамматики**. По сути, это набор правил для описания предложения. Формально грамматика G есть четверка { N, T , S, P } из четырех компонентов:

1) N - конечное множество нетерминальных символов,

2) T - конечное множество терминальных символов,

3) S - специальная цель или начало или известный символ ,

4) P - конечное множество правил и продукций.

Предложением является строка полностью состоящая из терминальных символов, выбранных из множества T. С другой стороны, множество N обозначает синтаксические классы грамматики, то есть общие компоненты или понятия, используемые в описании конструкции предложения.

Объединение множеств N и T обозначает словарный запас V грамматики - V=N U T и множества N и T не пересекаются N !U T=Ø

Английский язык с помощью этой четверки можно структурировать следующим образом: множество T здесь будет содержать 26 букв общего алфавита и знаки препинания. Множество N будет содержать синтаксические дескрипторы - простые, как существительное, прилагательное, глагол, а также более сложных, таких как словосочетания, наречного пункта и законченное предложение. Множество P будет содержать синтаксические правила, такие как описания словосочетания в виде последовательности прилагательного с последующим

существительным. Очевидно, что это набор может стать очень большим действительно - гораздо больше, чем T или даже N. Правило или продукция в сущности говорит нам, как мы можем получить предложение в языке. Мы начинаем с известных символов S, и, сделав последовательные замены, работая с последовательностью так называемых сентенциальных форм на пути к конечной строке, которая содержит только терминалы.

Эквивалентными грамматиками называются грамматики, описывающие один и тот же язык. Если предложение произведенное грамматикой имеет два или более AST деревьев, то грамматика называется неоднозначной. Некоторые потенциальные неоднозначности отосятся к контексту — такие грамматики называют контекстными. Напрмиер: «Time flies like an arrow» и «Fruit flies like a banana» - два предложения имеют одну конструкцию - «Noun Verb Adverbial phrase», но подсознательно второе предложение можно разобрать так: «Adjective Noun Verb Noun phrase»

Согласно Хомскому(1959), формальные грамматики делятся на четыре типа: **неограниченные, контекстно-зависимые, контекстно-свободные, регулярные**. Для отнесения грамматики к тому или иному типу необходимо соответствие всех её правил (продукций) некоторым схемам.

3.9 Классическая форма Бэкуса-Наура (BNF) и расширение Вирта (EBNF)

Существует множество способов указания правил, однако для описания более реалистичных грамматик, таких, которые могут быть использованы в определении языков программирования наиболее распространённым способом для описания правил на протяжении многих лет являлась **форма Бэкуса-Наура (англ. Backus-Normal-Form) — BNF.** Впервые эта форма блыла использована в презентации языка Algol 60

В классическом BNF, не-терминалу обычно дается подробное имя, и написано в угловые скобки, чтобы отличить его от терминального символа. Правила описываются так: leftside->definition

Здесь «->» можно интерпретировать как "определяется как" или "означает". В таких формах, как эта, левая сторона и определение состоит из строки, сцепленной с одним или более терминалами и нетерминалами. Для обозначаения множества альтернатива используется символ «|»

Рассмотрим пример небольшого подмножества Английского языка:

G = {N , T , S , P}

N = {<sentence> , <qualified noun> , <noun> , <pronoun> , <verb> , <adjective> }

T = { the , man , girl , boy , lecturer , he , she , drinks , sleeps , mystifies , tall , thin , thirsty }

S = <sentence>

P = { <sentence> → the <qualified noun> <verb> **(1)**

| <pronoun> <verb> **(2)**

<qualified noun> → <adjective> <noun> **(3)**

<noun> → man | girl | boy | lecturer **(4, 5, 6, 7)**

<pronoun> → he | she **(8, 9)**

<verb> → talks | listens | mystifies **(10, 11, 12)**

<adjective> → tall | thin | sleepy **(13, 14, 15)**

}

Множество правил определяет нетерминал <sentence> как состоящий из любого терминала "the", а затем <qualified noun> перед <verb>, или <pronoun> за которым следует <verb>.<qualified noun> представляет <adjective> последующим <noun> и <noun> является одним из терминальных символов "man" или "girl" или "boy" или "lecturer".<pronoun> - нетерминал "he" или "she", а <verb> это либо "talks" или "listens" или "mystifies". Здесь <sentence>, <noun>, <qualified noun>, <pronoun>, <adjective> и <verb> являются нетерминалы. Они не появляются в любом предложении языка.

Из грамматики, один нетерминал выделяется в качестве так называемой цели или стартового символа. Если мы хотим, сгенерировать произвольное предложение мы начинаем с символа цели и последовательно заменяем каждый нетерминал основываясь на правилах, определяющие, что этот нетерминал означает, пока все нетерминалы не быдут удалены. В приведенном выше примере символ <sentence> можно обозначить как симовл цели. Так, например, мы могли бы начать с <sentence> и от этого вывести сентенциальную форму:

the <qualified noun> <verb>

С точки зрения определений в последнем разделе мы говорим, что <sentence> непосредственно определяет "the <qualified noun> <verb>". Если теперь применить правило 3 (<qualified noun> → <adjective> <noun>) получим сентенциальную форму:

the <adjective> <noun> <verb>

С точки зрения определений в последнем разделе, "the <qualified noun> <verb>" непосредственно определяет "the <adjective> <noun> <verb>", в то время как <sentence> определил эту сентенциальную форму нетривиальным путем. Если теперь следовать этому, применяя правило 14 (<adjective> →thin) мы получаем форму:

the thin <noun> <verb>

Применяем правило 10(<verb> → talks) получаем:

the thin <noun> talks

И наконец применяем правило 6(<noun> → boy) получаем предложение:

the thin boy talks

Конечный результат удобно представлять в виде стуктурированного дерева разбора или дерево разбора, как на рисунке 3.9

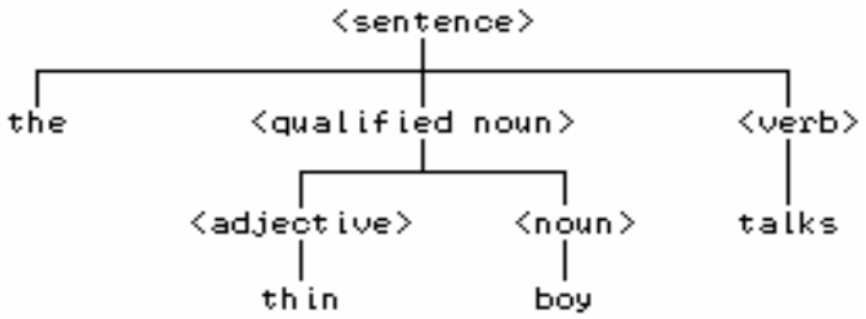

Рисунок 3.9 — Дерево разбора для предложения «the thin boy talks»

В этом представлении порядок в котором были использованы правила не очевидны, однако становится ясно, почему выражаются «терминал» и «нетерминал». В теории формальных языков, листья такого дерева — терминалы граматики, внутренние узлы — нетерминалы.

Существует множество возможных путек вывода из символа цели до окончательного предложения, в зависимости от порядка, в котором применяются правила. Левым каноническим выбором — называется порядок

выбора правила, в котором выберается самый левый нетерминал. Так же сущестует правый канонический выбор. Это очень важно не только для генерации но и при анализе типа - является ли существующий символ частью языка. Когда простое признание сопровождается определение базовой структуры дерева, мы говорим о разборе.

Попробуем разобрать предложение:

the thin boy talks

Начнем с цели <sentence>. Между правилами 1 и 2 выбираем 1, слева направо:

<sentence> → the <qualified noun> <verb>.

На данном этапе наше предложение «the thin boy talks» должно соответствовать форме the <qualified noun> <verb>. Теперь мы должны выбрать правила для замены <qualified noun> и <verb>. Опять же, выбираем правила слева направо. Тперь наше правило должно быть преобразовано в <adjective> <noun> <verb> используя правило 3.

Теперь наше предложение «the thin boy talks» должно соответствовать <adjective>

<noun> <verb>. Мы могли выбрать для замены любого из <adjective>, <noun> или <verb>, если мы читаем входную строку слева мы видим, что с помощью правила 14 мы можем уменьшить проблему соответствия остаточной входной строки "boy talks" в остаточной сентенциальной форме <noun> <verb>. И так далее.

Проблема разбора не всегда так легко решается, как в данном примере. Легко видеть, что алгоритмы, используемые для анализа предложения, чтобы увидеть, может ли оно быть получено из символа цели будет сильно отличаться от алгоритмов, которые могут быть использованы для создания предложений, начиная с начального символа. Методы, используемые для успешного разбора зависят весьма критически от порядка, в котором правила были указаны.

В BNF, правила могут определять нетерминалы рекурсивно. Например:

<qualified noun> → <noun> | <adjective> <qualified noun> (3a, 3b)

Существуют различные расширения описания включающие в себя BNF, например для описания рекурсий и похожих правил, могут использоваться специальные метасимволы, обозначающие использование конструкции множество раз. Расширения, введенные для упрощения таких

конструкций известны как **EBNF (Extended BNF).** Одной из популярных ворм EBNF является EBNF Вирта.

При определении Паскаль и Модула-2, Вирт подошел с одной из этих многих вариаций на BNF, который в настоящее время довольно широко используются(Вирт, 1977). Дальнейшие метасимволы используются, чтобы выразить более кратко многие ситуации, которые в противном случае потребовали бы комбинации операторов и Замыкание Клини. Обозначения для EBNF:

1. Нетерминалы — записываются в виде отдельных слов, как VarDeclaration

2. Терминалы — всё, что написано в кавычках, как «BEGIN» а не как сами по себе в BNF

3. «|» - так же как в BNF, для описания альтернативных правил

4. «( )» - скобки используются для обозначения группировки

5. «[ ]» - квадратные скобик, используются для обозначения дополнительной группы символов

6. «{ }» - фигурные скобки, используются для обозначения повторения группы символов

7. «=» - используется вместо ::= или →

8. «.» - используется для обозначения конца каждого правила

9. «(* *)» - используется для комментариев

10. «ε» - могут быть обработаны с помощью [] обозначения

11. «spaces» - незначительные по существу символы.

Пример:

EBNF = {Production}.

Production = nonterminal "=" Expression "." .

Expression = Term { "|" Term } .

Term = Factor { Factor } .

Factor = nonterminal | terminal | "[" Expression "]"

| "(" Expression ")" | "{" Expression "}" .

nonterminal = letter { letter } .

terminal ="'" character { character } "'" | '"' character { character } '"' .

character = (* определяется реализация *) .

Эффект в том, что нетерминалы менее "шумные", чем в более ранней форме BNF, в то время как терминалы "шумнее". Многие грамматики, используемые для определения языка программирования используют гораздо больше нетерминалов чем терминалов, так что это часто выгодно. Кроме того, поскольку терминалы и нетерминалы являются текстуально легко различимы, то обычно достаточно дать только набор правил P при записи грамматики, а не полную четверку {N, T, S, P}.

Разделение между синтаксисом и семантикой не всегда четко , т. к. это может обостриться в связи с тенденцией указать правила с использованием имен с четко семантическими обертонами, чьи сентенциальные формы уже отражают значения , которые будут прикреплены к приоритетам операторов. При указании семантики следует различать статическую семантику - то, что можно проверить во время компиляции, например требование, что нельзя перейти в середину процедуры, или что назначение может производиться только, если была произведена проверка типов, и динамическая семантика - особенности, которые действительно имеют значение только во время выполнения, например, эффекта постановки ветки на поток управления, или эффект от оператора присваивания по элементам хранения.

3.10 Детерминальный разбор сверху вниз. LL(k) парсинг.

Как обсуждалось в предыдущих разделах, задача Front End анализа — признание соответствия некоторого текста программы синтаксису и семантике языка. Это значит, что необходимо идентифицировать из готового предложения соответствия правилам и продукции языка. Сложность этого процесса зависит от сложности правил и продукции языка. Существует множество способов разбора предложений, однако мы сосредоточимся на довольно простом и в то же время эффективном методе известный как рекурсивный спуск или парсинг сверху вниз.

Для иллюстрации рассмотрим грамматику:

G = { N , T , S , P }

N = { A , B }

T = { x , y , z }

S = A

P =

A → xB **(1)**

B → z **(2)**

B → yB **(3)**

Попробуем разобрать предложение «xyyz», которое чётко состоит из терминалов этой грамматики. Начнём с символа цели и входной строки:

Sentential form S = A Input string xyyz

К сентенциальной форме A мы можем применить только правило 1:

Sentential form xB Input string xyyz

Найдено одно соответствие. Первый символ соответствует грамматике, пока всё в порядке. Можем его отбросить оставив «yyz»:

Sentential form B Input string yyz

В данном случае мы можем выбрать либо 2 либо 3 правило. Однако если присмотреться, то правило номер 3 очевидно:

Sentential form yB Input string yyz

Отсюда следует, что можно вывести «yz»

Sentential form B Input string yz

Сново применяем правило 3:

Sentential form yB Input string yz

Откуда следует, что можно вывести «z», после чего применив правило 2 и определить соответствие грамматике.

Метод, который мы используем является частным случаем так называемого **LL (k) разбора**. Терминология приходит из того, что мы сканируем входную строку слева направо (первый левый), применяя правила к крайнему левому нетерминалу в сентенциальной форме производим манипуляции (вторай левый), и, смотрим вперед на ближайшие K терминалы в строке ввода, чтобы решить, какое из правил применять на любом этапе. В нашем примере, довольно очевидно, к = 1; LL (1) разбор является наиболее распространенной формой LL (к) разбор на практике.

Не всегда удаётся разобрать строку так легко, как в примере выше. Иногда возникают конфликтные ситуации, неявные деревья и приходится производить так называемый откат.

Необходимо ввести понятия **терминальный стартовый символ** нетерминала, множество **FIRST(A)** - множество терминалов, с которых строки, полученные из A могут начинаться.

Определим FOLLOW(A) для нетерминала A как множество терминалов a, которые могут появиться непосредственно справа от A в некоторой сентенциальной форме грамматики, т.е. множество терминалов a таких, что существует вывод вида S *A a для некоторых , (N T)*. Заметим, что между A и a в процессе вывода могут находиться нетерминальные символы, из которых выводится e. Если A может быть самым правым символом некоторой сентенциальной формы, то $ также принадлежит FOLLOW(A).

null string problem, terminal successors

Грамматики, для которых таблица предсказывающего анализатора не имеет неоднозначно определенных входов, называются LL(1)-грамматиками.

4 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

4.1 Конструкция парсера и сканера

Для видов языка, удовлетворяющих правилам LL(1) грамматики описанные в предыдущих разделах, конструкция парсера оказывается очень простой. Синтаксис в этих языках регулируется правилами в форме нетерминал → допустимая строка, где допустимая строка представляет собой:

- осноновые символы и термины языка
- другие нетерминалы
- действия метасимволов, таких как {}, [] и |.

Мы выражаем эффект применения каждого правила, написав функцию в которой мы даем название нетерминала, который появляется слева.

Цель этой процедуры состоит в анализе последовательность символов, которая будет поставляться по запросу приходящего из сканера (лексический анализатор), и убедиться, что он имеет правильный формат, в противном случае выдать сообщения об ошибках. Т.е. Поиск токинов, проверка соответствия правилам, частичная генерация кода.

Такие анализаторы лучше реализовывать на языках, которые поддерживают ООП и рекурсию. C++ идеально подходит для этого. Прототипы классов позволяют описать сложные компоненты и скрыть внутреннюю организацию. Множество операций в C++ легко описать с помощью перегрузки. На диаграмме классов можно увидеть структуру парсера. Схема работы парсера изображена в приложении...

Сканер - более низкий уровень анализа. В некотором смысле, сканер или лексический анализатор можно рассматривать как просто еще один анализатор синтаксиса. Он обрабатывает грамматику с правил, связанных нетерминалами таких как идентификатор, поставляемых, в сущности, как отдельные символы исходного текста. При использовании в сочетании с анализатором высшего уровня возникает тонкое смещение акцентов. Каждый вызов сканера выполняет очень много подъемов вверх, а не сверху вниз; его задача заканчивается, когда она сократила строку символов. Эти маркеры или нетерминалы затем рассматриваются в качестве терминалов на более высоком уровне рекурсивного спуска, который анализирует структуру фразы блока, заявление, слова и так далее.

Есть, по крайней мере, пять причин чтобы отделить сканер от главного парсера:

1. Продукции, участвующие как правило, очень просты. Очень часто они составляют регулярные выражения, а затем сканер может быть запрограммирован, не прибегая к методам, как рекурсивного спуска.

2. Символ как идентификатор лексически эквивалентен "зарезервированному слову». Различие может быть разумно, как только основной токен был синтезирован.

3. Набор символов может изменяться от машины к машине, вариации легко изолирован в этой фазе.

4. Семантический анализ числовой константы легко выполняется параллельно с лексическим анализом.

5. На сканер может быть возложена ответственность за отсеивая лишних разделителей, как пробелы и комментарии, которые редко интересны к формулировке более высокого уровня грамматики.

Сканер обязательно парсит сверху вниз, и для простоты реализации желательно, чтобы продукции, определяющие токены грамматики также подчиняться LL(1) правилам.

Есть две основные стратегии, которые используются в строительстве сканера:

1. Вместо того, чтобы разложить на набор рекурсивных подпрограмм, простые сканеры часто пишутся в разовой основе, контролируемой большим CASE switch заявлениями выбирая токены, которые иногда отличаются на основе их начальных символов.

2. С другой стороны, так как они, как правило, считывают число символов, сканеры часто пишутся в виде конечного автомата (finite state automaton FSA), контролируемые циклом, на каждой итерации которого один символ поглощается, машина движется между несколькими из "состояний", определяющее символ. Этот подход имеет то преимущество, что строительство можно сформулировать в терминах широким развитием теории автоматов, что привело к алгоритмам, из которых сканер и генераторы могут быть построены автоматически.

Мы используем первый случай, Coco/R — второй. Схема работы сканера изображена в приложении …

4.2 Синтаксически управляемая трансляция

Основная цель парсеров - это признание или отказ от входных строк которые утверждают, что являются предложениями рассматриваемого языка . Рассмотрим простой пример, который поможет кристаллизовать концепцию парсера данного дипломного проекта. Обратимся вновь к грамматике, которая может описать простые алгебраические выражения, с помощью которой можно обрабатывать выражения в скобках в дополнение к обычным четырем операторам:

Expression = Term { "+" Term | "-" Term } .

Term = Factor { "*" Factor | "/" Factor } .

Factor = identifier | number | "(" Expression ")" .

Легко проверить, что это грамматика относится к LL (1). Простой рекурсивный спуск парсера легко сконструирован с целью принятия допустимого выражения входного символа или выполнения прерывания с соответствующим сообщением об ошибке, если входное выражение имеет неверный формат.

```
void Expression(void); // function prototype

void Factor(void)

// Factor = identifier | number | "(" Expression ")" .

{

    switch (SYM.sym)

    {

        case identifier:

        case number:

            getsym();

            break;

        case lparen:

            getsym();

            Expression();
```

```
                accept(rparen, " Error - ')' expected");
                break;
            default:
                printf("Unexpected symbol\n");
                exit(1);
        }
}
void Term(void)
// Term = Factor { "*" Factor | "/" Factor } .
{
        Factor();
        while (SYM.sym == times || SYM.sym == slash)
        {
            getsym();
            Factor();
        }
}
void Expression(void)
// Expression = Term { "+" Term | "-" Term } .
{
        Term();
        while (SYM.sym == plus || SYM.sym == minus)
        {
            getsym();
            Term();
```

```
        }
}
```

Обратим внимание, что в этом примере мы предполагали существование сканера более низкого уровня, который признает фундаментальные терминальные символы и создает глобально доступную переменную SYM, который имеет структуру:

```
enum symtypes {
        unknown, eofsym, identifier, number, plus, minus, times, slash,
        lparen, rparen, equals
};
struct symbols {
        symtypes sym; // class
        char name; // lexeme
        int num;  // value
};
symbols SYM; // Source token
```

Синтаксический анализатор надлежащее требует, чтобы первоначальным вызовом был getsym () перед вызовом Expression() в первый раз.

Так же мы предположили существования обработчика ошибок, который будет описан чуть позже.

Теперь рассмотрим вопрос о чтении допустимой строки на этом языке, и перевод его в строку, которая имеет тот же смысл, но которая выражается в Postfix обозначение (обратная польская запись). При этом операторы следуют попарно операнам, и нет никакой необходимости в скобках. Например, выражение: (a+b)*(c-d) должно быть переведено в ab+cd-*

Мы читаем входную строку слева направо и сразу копируем все операнды в выходной поток, как только они признаются, но мы задерживаем копирования операторов пока мы не можем сделать это в порядке, который относится к правилам приоритета операций. Это хорошо известная проблема, которая решается с помощью обратной польской записи.

```
void Factor(void)
// Factor = identifier | number | "(" Expression ")" .
{
        switch (SYM.sym)
        {
                case identifier:
                case number:
                        printf(" %c ", SYM.name);
                        getsym();
                        break;
                case lparen:
                        getsym();
                        Expression();
                        accept(rparen, " Error - ')' expected");
                        break;
                default:
                        printf("Unexpected symbol\n");
                        exit(1);
        }
}
void Term(void)
// Term = Factor { "*" Factor | "/" Factor } .
{
        Factor();
        while (SYM.sym == times || SYM.sym == slash)
```

```
        {
                switch (SYM.sym)
                {
                        case times:
                                getsym();
                                Factor();
                                printf(" * ");
                                break;
                        case slash:
                                getsym();
                                Factor();
                                printf(" / ");
                                break;
                }
        }
}
void Expression(void)
// Expression = Term { "+" Term | "-" Term } .
{
        Term();
        while (SYM.sym == plus || SYM.sym == minus)
        {
                switch (SYM.sym)
                {
                        case plus:
```

```
                    getsym();
                    Term();
                    printf(" + ");
                    break;
                case minus:
                    getsym();
                    Term();
                    printf(" - ");
                    break;
            }
        }
}
```

В данном примере мы легко перешли от синтаксического анализа к компиляции. Мы проиллюстрировали простой пример синтаксически управляемой программы - той, в которой основной алгоритм легко разработан из понимания основной синтаксической структуры. Однако данный пример не достаточно подходит для обработки многих ситуаций, в которых требуется понимание более глубокого смысла.

Контекстно-свободная грамматика может быть использована для описания многих особенностей языков программирования. Такие грамматики эффективно определяют вывод или дерево разбора для каждой синтаксически правильной программы на языке, и мы видели, что с осторожностью можно построить грамматику так, чтобы дерево разбора в некотором роде отражало смысл программы.

Несмотря на какие-либо явные практические трудности, наши представления о формальной грамматике может быть расширено, чтобы попытаться захватить сущность атрибутов, связанных с узлами, за счет расширения обозначения.

Немного дополним дерево разбора в предыдущем примере атрибутами. Необходимо отметить, что информация всегда передается "вверх" дерева, или "вне" соответствующих процедур. Т.е. параметры должны быть переданы "по ссылке".

```
void Factor(int &value)
// Factor = identifier | number | "(" Expression ")" .
{
	switch (SYM.sym)
	{
		case identifier:
		case number:
			value = SYM.num; getsym(); break;
		case lparen:
			getsym();
			Expression(value);
			accept(rparen, " Error - ')' expected");
			break;
		default:
			printf("Unexpected symbol\n");
			exit(1);
	}
}
void Term(int &value)
// Term = Factor { "*" Factor | "/" Factor } .
{
	int factorvalue;
	Factor(value);
	while (SYM.sym == times || SYM.sym == slash)
```

```
        {
                switch (SYM.sym)
                {
                        case times:
                                getsym();
                                Factor(factorvalue);
                                value *= factorvalue;
                                break;
                        case slash:
                                getsym();
                                Factor(factorvalue);
                                value /= factorvalue; break;
                }
        }
}
void Expression(int &value)
// Expression = Term { "+" Term | "-" Term } .
{
        int termvalue;
        Term(value);
        while (SYM.sym == plus || SYM.sym == minus)
        {
                switch (SYM.sym)
                {
                        case plus:
```

```
                    getsym();
                    Term(termvalue);
                    value += termvalue;
                    break;
                case minus:
                    getsym();
                    Term(termvalue);
                    value -= termvalue;
                    break;
            }
        }
}
```

Атрибуты не всегда поднимаются по дереву. Пример небольшой программы на языке, который мы описали в данном дипломном проекте:

```
PROGRAM Silly;
    CONST
        Bonus = 4;
    VAR
        Pay;
BEGIN
    WRITE(Pay + Bonus)
END.
```

Структура дерева разбора изображена на рисунке 4.1

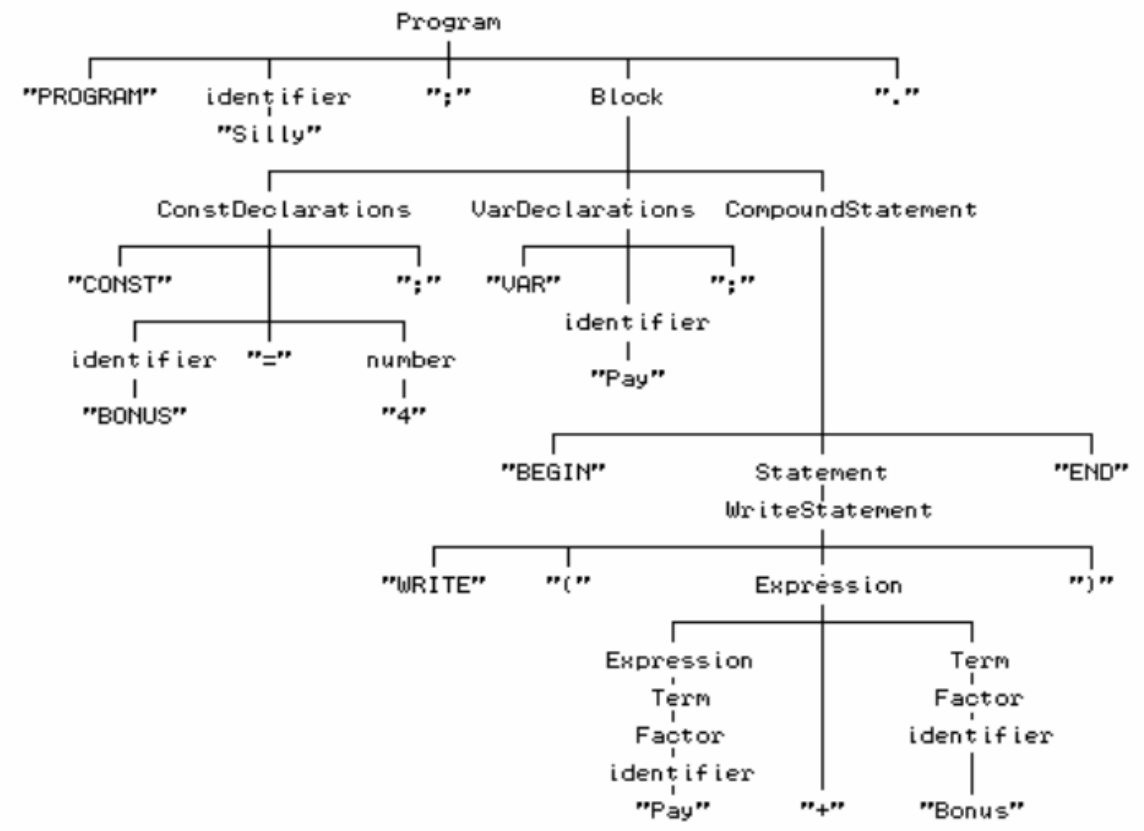

Рисунок 4.1 — Дерево разбора для небольшой программы

В этом случае мы можем определить булевой IsConstant и IsVariable атрибуты CONST узлов и VAR которые передаются вверх по дереву, а уже потом передаются обратно вниз и наследуются другими узлами, как Bonus и Pay (см. рисунок 4.2).

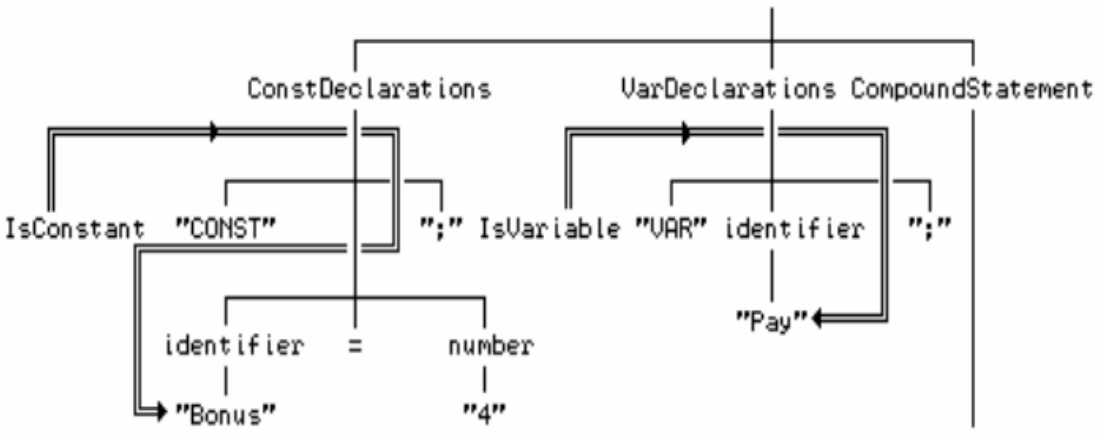

Рисунок 4.2 — Атрибуты, передаваемые вверх и вниз по дереву разбора

Конечно это должно происхоить намного позже. Атрибуты как правило являются частью окружающей среды(англ. Environment). Компиляция или разбор программ обычно начинается в «стандартной» среде, в которую включены идентификаторы, такие как TRUE, FALSE, ORD, CHR и так далее. Эта среда наследуется программой, затем блок, а затем ConstDeclarations, которая расширяет среду и передает её обратно вверх по наследству в расширенной форме по VarDeclarations которая расширяет её дальше и передает назад, так что она может затем быть передана к CompoundStatement. Данная схема изображена на рисунке 4.3

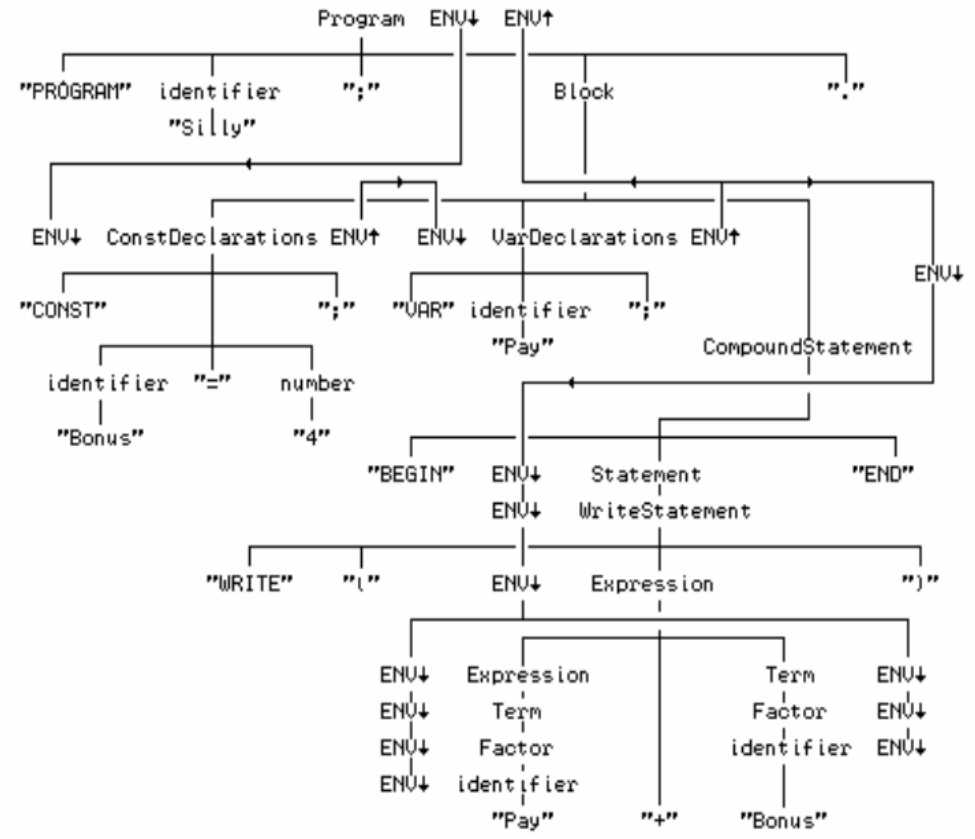

Рисунок 4.3 — Модификация среды разбора

4.3 Front End фазы

В разделе моделирования программного средства мы прокомментировали, что компилятор часто развивается как последовательность фаз, из которых синтаксический анализ является только одной из них. Кроме рекурсивного спуска парсера есть ещё другие фазы. Общая структура компилятора, разрабатываемого в данном дипломном проекте изображена на рисунке 4.4.

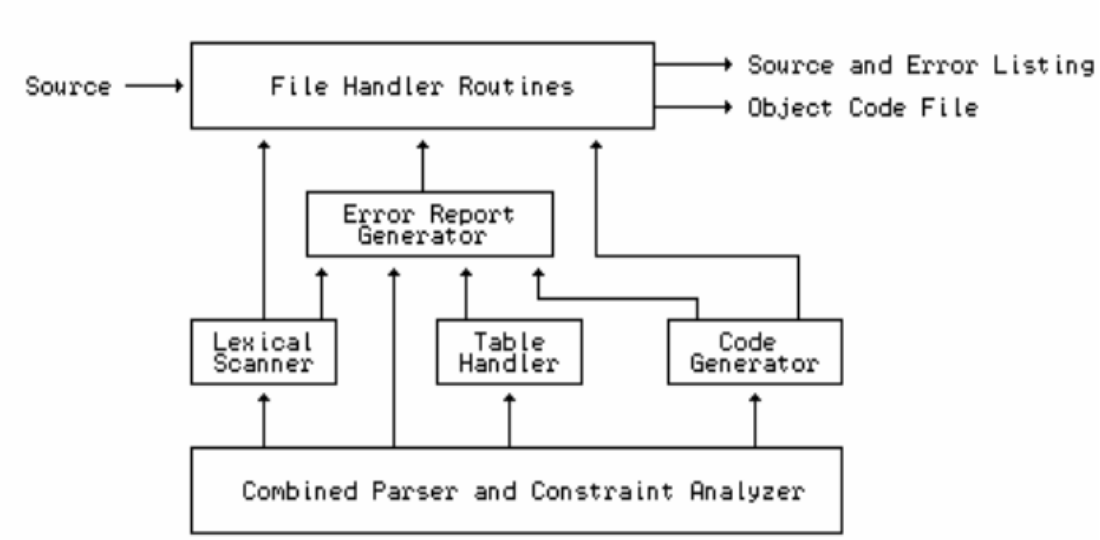

Рисунок 4.4 — Отношения главных компонентов простого компилятора

Подчеркиваю, что фазы не обязательно должны быть последовательными. В компиляторах рекурсивного спуска фазы синтаксического анализа, семантического анализа и генерации кода очень часто чередуются, особенно если язык источник сконструирован таким образом, чтобы обеспечить один проход компиляции. Тем не менее , полезно думать о разработке модульных компонентов для обработки различных этапов, с использованием четких простых интерфейсов между ними.

В принципе, основная подпрограмма нашего компилятора должна напоминать что-то вроде следующего:

```
void main(int argc, char *argv[])

{

    char SourceName[256], ListName[256];

    // handle command line parameters

    strcpy(SourceName, argv[1]);

    if (argc > 2) strcpy(ListName, argv[2]);

    else appendextension(SourceName, ".lst", ListName);

    // instantiate compiler components
```

```
    SRCE  *Source = new SRCE(SourceName, ListName, "Compiler Version
1", true);

    REPORT *Report = new REPORT(Source);

    SCAN  *Scanner = new SCAN(Source, Report);

    CGEN  *Cgen = new CGEN(Report);

    TABLE *Table = new TABLE(Report);

    PARSER *Parser = new PARSER(CGen, Scanner, Table, Report);

    // start compilation

    Parser->parse();

}
```

Из этого видно, что экземпляры различных классов строятся динамически, и что их конструкторы установают связи между ними, которые соответствуют тем, которые показаны на рисунке 4.4.

4.3.1 Обработчик источника

Среди процедур работы с файлами можно найти ту, которая имеет задачу передачи источника, символ за символом, к сканеру или лексическому анализатору (который собирает его в символы для последующего разбора анализатором синтаксиса) — **обработчик источника** (англ. Source handling). В идеале этот исходный обработчик сканирует текст программы только один раз, от начала до конца, и в однопроходных компиляторах это всегда должно быть возможным.

Интерфейс необходимый между исходным обработчиком и лексическим анализатором является простым, и может быть представлена подпрограммой, которая просто извлекает следующий символ из источника каждый раз когда она вызывается. Можно создать пакет с подпрограммами, которые предполагают ответственность за производство листинга исходного кода, и, при необходимости, производить листинг сообщения об ошибке, потому что все эти требования будут зависеть от устройство ввода / вывода (консоль, файл). Это позволяет добавить некоторые дополнительные функциональные возможности, и, таким образом, общий интерфейс к нашему классу обработки источника определяется:

```
class SRCE {

    public:
```

```
        FILE *lst; // listing file
        char ch; // latest character read
        void nextch(void);
        // Returns ch as the next character on this source line, reading a new
        // line where necessary. ch is returned as NUL if src is exhausted.
        bool endline(void);
        // Returns true when end of current line has been reached
        void listingon(void);
        // Requests source to be listed as it is read
        void listingoff(void);
        // Requests source not to be listed as it is read
        void reporterror(int errorcode);
        // Points out error identified by errorcode with suitable message
        virtual void startnewline() {;}
        // Called at start of each line
        int getline(void);
        // Returns current line number
        SRCE(char *sourcename, char *listname, char *version, bool
listwanted);
        // Opens src and lst files using given names.
        // Resets internal state in readiness for starting to scan.
        // Notes whether listwanted. Displays version information on lst file.
        ~SRCE();
        // Closes src and lst files
};
```

Некоторые аспекты нуждаются в комментировании:

- Здесь не отображены приватные свойства класса, полную версию можно увидеть в диаграмме классов и листинге программы
- Процедура startnewline была объявлена виртуальной, так что простой класс может обеспечить добавление дополнительного материала в начале каждой новой линии - например, номера строк или адресами объектного кода.
- В идеале, оба источника и выдача файлы должны оставаться закрытыми. Этот обработчик источник заявляет исходный файл открытым только потому, что мы можем добавить метку или отладочную информацию в то время как система разрабатывается.
- Конструктор и деструктор класса берут на себя ответственность за открытие и закрытие файлов, чьи имена передаются в качестве аргументов.
- Источник сканируется и отражает всю линию за один раз, так как это делает последующее сообщения об ошибках гораздо проще.
- Обработчик вставляет дополнительный пробел в конце каждой строки. Это отделяет остальные фазы системы от конфликтов и также гарантирует, что ни один символ не может распространяться на разрывах строк.
- Нет возможности читать после конца файла - попытки сделать это просто вернут пустой NULL символ.
- Процедура ReportError не будет отображать сообщения об ошибках, если не будут отсканированы минимальное число символов. Это помогает подавить каскад сообщений об ошибках, которые могли бы появиться в любой точке во время восстановления после ошибок.
- Используется STDIO библиотека, а не IOStreams, главным образом для использования краткой формы printf.

4.3.2 Обработчик ошибок

Как видно из рис 4.4, большинство компонентов компилятора должны быть готовы, чтобы просигнализировать, что что-то пошло наперекосяк в процессе компиляции. Чтобы это произошло в едином порядке, можно ввести базовый класс с очень маленькой интерфейсом:

```
class REPORT {
    public:
        REPORT();
        // Initializes error reporter
        virtual void error(int errorcode);
        // Reports on error designated by suitable errorcode number
        bool anyerrors(void);
        // Returns true if any errors have been reported
    protected:
        bool errors;
};
```

Когда ошибка обнаружена, передавём уникальный номер, чтобы отличить ошибку. Поскольку метод error является виртуальным, можно легко получить более подходящий класс для того, чтобы изменить любую другую часть системы. Для нашей системы мы можем сделать это следующим образом:

```
class langReport : public REPORT {
    public:
        langReport(SRCE *S)
        { Srce = S; }
        virtual void error(int errorcode)
        { Srce->reporterror(errorcode); errors = true; }
    private:
        SRCE *Srce;
};
```

4.3.3 Лексический анализатор

Основной задачей сканера является предоставления какого-то способа уникальной идентификации каждого последующего токена или символа в исходном коде, который компилируется.

Интерфейс между сканером и главным парсером удобно обеспечивается обычным getsym для возвращения параметр SYM структурного типа, собранного из исходного текста. Это может быть достигнуто путем определения класса с общего интерфейса следующим образом:

```
enum SCAN_symtypes {

    SCAN_unknown, SCAN_becomes, SCAN_lbracket, SCAN_times,
    SCAN_slash, SCAN_plus,SCAN_minus, SCAN_eqlsym, SCAN_neqsym,
    SCAN_lsssym, SCAN_leqsym, SCAN_gtrsym,SCAN_geqsym,
    SCAN_thensym, SCAN_dosym, SCAN_rbracket, SCAN_rparen,
    SCAN_comma, SCAN_lparen, SCAN_number, SCAN_stringsym,
    SCAN_identifier, SCAN_coendsym, SCAN_endsym, SCAN_ifsym,
    SCAN_whilesym, SCAN_stacksym, SCAN_readsym, SCAN_writesym,
    SCAN_returnsym, SCAN_cobegsym, SCAN_waitsym, SCAN_signalsym,
    SCAN_semicolon, SCAN_beginsym, SCAN_constsym, SCAN_varsym,
    SCAN_procsym, SCAN_funcsym, SCAN_period, SCAN_progsym,
    SCAN_eofsym

};

const int lexlength = 128;

typedef char lexeme[lexlength + 1];

struct SCAN_symbols {

    SCAN_symtypes sym; // symbol type

    int num; // value

    lexeme name; // lexeme

};

class SCAN {

    public:

        void getsym(SCAN_symbols &SYM);
```

```
        // Obtains the next symbol in the source text

        SCAN(SRCE *S, REPORT *R);

        // Initializes scanner

};
```

Некоторые аспекты этого интерфейса заслуживают комментариев:

- SCAN_symbols предусматривает возвращение не только уникального типа символа, но и соответствующий текстовое представление (известный как лексемы), а также числовое значение, когда символ признается как число.
- SCAN_unknown обслуживает ошибочных символов, как # и? который на самом деле не являются частью терминального алфавита. Вместо того, чтобы принять меры, простой сканер возвращает символ без комментариев, и оставляет парсеру, чтобы разобраться. Точно так же, явное SCAN_eofsym всегда возвращается, если getsym вызывается после конца исходного кода.
- Порядок перечисления SCAN_symtypes является значительным, и поддерживает интересную форму восстановления после ошибок, что будет обсуждаться далее
- По мере продвижения разбора фаз перечисление будет расширяться.

4.3.4 Синтаксический анализатор

Открытый интерфейс парсера может быть очень простым:

```
class PARSER {

    public:

        PARSER(SCAN *S, REPORT *R);

        // Initializes parser

        void parse(void);

        // Parses the source code

};
```

где мы отмечаем, что конструктор класса связывает экземпляр парсера с соответствующими экземплярами сканера и обработчика ошибок. Однако в полной версии данный интерфейс будет расширен, т. к. здесь необходимо добавить генерайию Р-кода.

Для того, чтобы обеспечить рекурсивный спуск парсера, после обнаружения синтаксической ошибки, т. е. востановления — необходимо объявить дополнительные методы:

```
void synchronize(SCAN_symtypes SmallestElement, int errorcode)

{

    if (SYM.sym >= SmallestElement) return;

    reporterror(errorcode);

    do { getsym(); } while (SYM.sym < SmallestElement);

}
```

Таким образом:

```
void Statement(void)

    // Statement = [ CompoundStatement | Assignment | IfStatement

    //      | WhileStatement | WriteStatement | ReadStatement ] .

{

    synchronize(SCAN_identifier, 15);

    // We shall return correctly if SYM.sym is a semicolon or END (empty statement)

    // or if we have synchronized (prematurely) on a symbol that really follows

    // a Block

    switch (SYM.sym)

    {

        case SCAN_identifier: Assignment(); break;

        case SCAN_ifsym: IfStatement(); break;
```

```
            case SCAN_whilesym: WhileStatement(); break;
            case SCAN_writesym: WriteStatement(); break;
            case SCAN_readsym: ReadStatement(); break;
            case SCAN_beginsym: CompoundStatement(); break;
            default:
                return;
        }
        synchronize(SCAN_endsym, 32);
        // In some situations we shall have synchronized on a symbol that can start
        // a further Statement, but this should be handled correctly from the call
        // made to Statement from within CompoundStatement
}
```

Так же нам понадобится:

```
bool inFirstStatement(SCAN_symtypes Sym)
// Returns true if Sym can start a Statement
{ return (Sym == SCAN_identifier || Sym == SCAN_beginsym ||
        Sym >= SCAN_ifsym && Sym <= SCAN_signalsym);
}
```

Символы не всегда падают однозначно в FIRST или FOLLOW последовательность, а в крупных языков может быть несколько ключевых слов (например, END, CASE и OF), которые могут появляться в самых различных контекстах. Если новые ключевые слова будут добавлены в развивающемся языке, большое внимание должно быть уделено, чтобы поддерживать оптимальный порядок; если неуместно значение токена, система анализа ошибок может сильно пострадать.

Полная реализация парсера содержится в листинге.

4.3.5 Обработка таблицы символов

В предыдущих главах утверждалось, что было бы выгодно, чтобы разделить наш компилятор на отдельные фазы для анализа синтаксиса и генерации кода. Одним из хороших поводов для этого является то, что нужно изолировать машиннозависимую часть компиляции, насколько это возможно на основе анализа языка.

Конечно, любой интерфейс между источником и объектным кодом должен принять во внимание понятия, связанных с данными, как хранения, адреса и представления данных, а также управления, связанных с них, как расположение счетчика, последовательного выполнения и команды перехода, которые являются основополагающими для почти всех машин. Как правило, машины позволяют некоторые операции, которые имитируют арифметические или логические операции над битами данных моделей, которые имитируют цифры или символы, эти модели хранятся в массиве, как структура памяти, элементы которой отличаются по адресам. В языках высокого уровня этим адресам обычно даются мнемонические имена. Контекстно-свободной синтаксис многих языков высокого уровня, как это происходит, редко проводит различие между "адресом" для переменной и "значением", связанного с этой переменной. Отсюда находим такие заявления

X : = X + 4

в которой «X» слева является оператором «:=» фактически представляет собой адрес, (иногда называют L- значение X), а X справа (иногда называемый R- значение X) фактически представляет собой значение количества в настоящее время расположенному по тому же адресу. Если мы свяжем это обратно в постановках, используемых в нашей грамматике , мы обнаружим, что каждый X в приведенном выше назначении был синтаксически номенклатурный . Семантически эти два обозначения очень разные - мы должны обращаться к одному, который представляет адрес в качестве переменной обозначения, и на то, что представляет собой значение как Value.

Для выполнения этой задачи, интерфейс генерации кода потребует извлечение дополнительной информации, связанной с пользовательскими идентификаторами и лучше всего хранить это в таблице символов. В случае постоянных, необходимо записать туда соответствующие значения, и в случае переменных нужно записать, связанные адреса и требования хранения (элементы массива переменных будет занимать непрерывный блок памяти). Если мы можем предположить, что наша машина оборудована моделью

памяти - «линейным массив», эта информация легко добавляется так, как объявлены переменные.

Обработка различные видов записей, которые должны быть сохранены в таблице символов может быть сделано различными способами. В реализации на основе классов в объектно-ориентированном можно определить абстрактный базовый класс для представления базового типа записи, а затем наследовать классы от этого и представлять записи для переменных или констант (и, со временем, записи, процедуры, классы и любые другие формы записи, которые, потребуются). Продолжим декларацию TABLE_entries:

```
struct TABLE_entries {
        TABLE_alfa name; // identifier
        TABLE_idclasses idclass; // class
        union {
                struct {
                        int value;
                } c; // constants
                struct {
                        int size, offset; // number of words, relative address
                        bool scalar; // distinguish arrays
                } v; // variables
        };
};
```

В листинге программы дипломмного проекта можно найти полную реализацию таблицы символов в виде массива фиксированной длинны.

4.4 Back End фазы

После того, как Front End фазы отработали проанализировали исходный код, наступает очередь Back End фаз. В основном это синтез объектного кода. Генерация кода, любой формы, означает, что мы рассматриваем семантику нашего языка и нашу целевую машину, и взаимодействие между ними, более подробно.

4.4.1 Интерфейс генерации кода

При рассмотрении интерфейса между анализом и генерацией кода основная цель - стремиться в некоторой степени достичь независимости машины. Генерация кода должна происходить без излишнего анализа . Обычная техника для достижения этой, казалось бы, невыполнимой задачи, является определение гипотетическиой машины, с набором инструкций и архитектуры в удобном для исполнения программ на языке оригинала, но, не будучи слишком далеко от реальной системы, для которой требуется компилятор. Действие процедуры интерфейса будет выполнять трансляцию исходной программы в эквивалентную последовательность операций для гипотетической машины. Вызов этих процедур могут быть встроены в анализаторе без чрезмерной заботы о том, как окончательный генератор переведёт операции в объектный код для целевой машины. Действительно, как мы уже упоминали, некоторые операции никогда не производят реальный машинный код.

Интерфейс генерации кода может принимает следующую форму:

```
enum CGEN_operators {
        CGEN_opadd, CGEN_opsub, CGEN_opmul, CGEN_opdvd, CGEN_opeql,
        CGEN_opneq, CGEN_oplss, CGEN_opgeq, CGEN_opgtr, CGEN_opleq
};
typedef short CGEN_labels;
class CGEN {
        public:
                CGEN_labels undefined;
                // for forward references
                CGEN(REPORT *R);
```

```
        // Initializes code generatorvoid negateinteger(void);
        // Generates code to negate integer value on top of evaluation stack
        void binaryintegerop(CGEN_operators op);
        // Generates code to pop two values A,B from stack and push value A op B
        void comparison(CGEN_operators op);
        // Generates code to pop two values A,B from stack; push Boolean value A op B
        void readvalue(void);
        // Generates code to read an integer; store on address found on top-of-stack
        void writevalue(void);
        // Generates code to pop and then output the value at top-of-stack
        void newline(void);
        // Generates code to output line mark
        void writestring(CGEN_labels location);
        // Generates code to output string stored from known location
        void stackstring(char *str, CGEN_labels &location);
        // Stores str in literal pool in memory and returns its location
        void stackconstant(int number);
        // Generates code to push number onto evaluation stack
        void stackaddress(int offset);
        // Generates code to push address for known offset onto evaluation stack
        void subscript(void);
        // Generates code to index an array and check that bounds are not exceeded
```

```
    void dereference(void);
    // Generates code to replace top-of-stack by the value stored at the
    // address currently stored at top-of-stack
    void assign(void);
    // Generates code to store value currently on top-of-stack on the
    // address stored at next-to-top, popping these two elements
    void openstackframe(int size);
    // Generates code to reserve space for size variables
    void leaveprogram(void);
    // Generates code needed as a program terminates (halt)
    void storelabel(CGEN_labels &location);
    // Stores address of next instruction in location for use in backpatching
    void jump(CGEN_labels &here, CGEN_labels destination);
    // Generates unconditional branch from here to destination
    void jumponfalse(CGEN_labels &here, CGEN_labels destination);
    // Generates branch from here to destination, conditional on the
Boolean
    // value currently on top of the evaluation stack, popping this value
    void backpatch(CGEN_labels location);
    // Stores the current location counter as the address field of the branch
    // instruction assumed to be held in an incomplete form at location
    void dump(void);
    // Generates code to dump the current state of the evaluation stack
    void getsize(int &codelength, int &initsp);
    // Returns length of generated code and initial stack pointer
```

```
        int gettop(void);

        // Returns the current location counter
};
```

Некоторые пояснения:

- Процедурам генерации кода были даны имена, и можно предположить, что они на самом деле выполнять такие операции, как jump. Но на самом деле они только генерируют код для таких операций.
- Существует неизбежное взаимодействие между этим классом и классом эмулируемой машины, для комманд, код которых должен быть сгенерирован — в реализации придется импортировать тип машиного адреса, поэтому будет целесообразным экспортировать процедуру (getsize), которая позволит компилятору определить количество сформированного кода.
- Код для манипулирования с данными на такой машине может быть создан с помощью вызовова процедур, таких как stackconstant, stackaddress, stackstring, subscript и dereference для доступа к хранилищу. Вызов процедур negateinteger и binaryintegerop предназначен для генерации кода для выполнения простых арифметических операций. И наконец вызов assign генерирует присваивание. Например для кода A := 4 + List[5] где List имеет 14 элементов будет вызвано следующее:

```
stackaddress(offset of A)
        stackconstant(4)
        stackaddress(offset of List[0])
        stackconstant(5)
        stackconstant(14)
        subscript
        dereference
        binaryintegerop(CGEN_opadd)
assign
```

- Адрес связанный с массивом в таблице символов будет обозначать смещение первого элемента массива во время выполнения.
- Для генерации кода исользуются оперции ввода/вывода readvalue,

writevalue, writestring и newline.

- Для операций сравнения используется вызов comparison
- Управляющие комманды более интерестны, т. к. нужно учитывать порядок, в котором будет выполнен. Например исходный код

    IF Condition THEN Statement

будет сегнерирован следующий код:

    code for Condition

    IF NOT Condition THEN GOTO LAB END

    code for Statement

    LAB continue

и проблема в том, что когда мы доберемся до стадии генерации GOTO LAB мы не знаем адрес, который будет применяться к LAB.

То же самое здесь:

    WHILE Condition DO Statement

генерируем код:

    LAB code for Condition

        IF NOT Condition THEN GOTO EXIT END

        code for Statement

        GOTO LAB

    EXIT continue

Здесь мы должны знать адресс LAB когда мы начниаем генерировать код для Condition, но мы не будем знать адресс EXIT, когда вызываем GOTO EXIT.

В целом решение этой проблемы может потребоваться с использованиемдвупроходной системы. Но тем не менее мы будем использовать один проход с динамической модификацией адресов. Генерируем команды перехода с помощью jump(here, label) и jumponfalse(here, label) и введём две вспомогательные процедуры: storelabel(location) и backpatch(location) чтобы помнить местоположение инструкции, и, чтобы иметь возможность восстановить адресные поля полностью сформированных команд перехода на более позднем этапе. Генератор кода экспортирует особое значение типа CGEN_labels которые могут быть использованы для создания временного целевого назначения для таких неполных инструкций.

- Побочным эффектом в WhileStatement и IfStatement является то, что мы не можем использовать явные метки. Метки могут быть обработаны, объявив соответствующие переменные локальными для подпрограммы синтаксического анализа, как IfStatement. Каждый раз, когда рекурсивный вызов делается на IfStatement — объявляются новые переменные и существуют, пока вся конструкция не будет разобрана. Затем они удаляются. При компиляции IfStatement используется следующая техника:

```
void IfStatement(void)

// IfStatement = "IF" Condition "THEN" Statement .

{

        CGEN_labels testlabel;

        // must be declared locally

        getsym();

        // scan past IF

        Condition();

        // generates code to evaluate Condition

        jumponfalse(testlabel,undefined); // remember address of incomplete instruction

        accept(thensym);

        // scan past THEN
```

```
        Statement();

        // generates code for intervening Statement(s)

        backpatch(testlabel);

        // use local test value stored by jumponfalse

}
```

- Возможно понадобится специальная уборка кода, когда мы будем покидать основной блок. Этот код описыватеся в openstackframe и leaveprogram

- gettop - источник может предоставить подробную информацию об адресах объектного кода, соответствующих заявлений в источнике.

4.4.2 Генерация кода для стековой машины

Реализация генератора кода устроена по принципу генерации кода на лету. Искохдный код можно посмотреть в листинге программы дипломного проекта. Основные моменты, на которые стоит обратить внимание:

- Внешний экземпляр класса STKMC доджен быть видимым генератором кода. По мере генерации кода он хранится непосредственно в массиве кода Machine->mem.

- Конструктор класса генератора кода инициализирует два частных члена - расположение счетчика (codetop), необходимых для хранения команд и лучшие указатели памяти (stktop) необходимо для хранения строковых литералов.

- Процедура stackaddress передает простое значение адрес в таблице символов, и преобразует это в смещение, которое позже будет вычисляться по отношению к cpu.bp когда программа будет выполнятся.

- Генератор кода подавляет дальнейшие попытки сгенерировать код, если происходит переполнение памяти, хотя и предоставляет синтаксический разбор, чтобы продолжить.

Проиллюстрируем сгенерированный объектный код для простой программы:

```
0: PROGRAM Debug;
0:      CONST
0:              VotingAge = 18;
0:      VAR
0:              Eligible, Voters[100], Age, Total;
0:      BEGIN
2:              Total := 0;
7:              Eligible := 0;
12:             READ(Age);
15:             WHILE Age > 0 DO
23:             BEGIN
23:                     IF Age > VotingAge THEN
29:                     BEGIN
31:                             Voters[Eligible] := Age;
43:                             Eligible := Eligible + 1;
52:                             Total := Total + Voters[Eligible -1];
67:                     END;
71:                     READ(Age);
74:             END;
76:             WRITE(Eligible, ’ voters. Average age = ’, Total / Eligible);
91:     END.
```

Таблица символов будет иметь записи:

```
1 DEBUG Program
2 VOTINGAGE Constant 18
3 ELIGIBLE Variable 1
```

4 VOTERS Variable 2

5 AGE Variable 103

6 TOTAL Variable 104

и сгенерированный код примет следующий вид:

0 DSP 104 Reserve variable space

2 ADR -104 address of Total

4 LIT 0 push 0

6 STO Total := 0

7 ADR -1 address of Eligible

9 LIT 0 push 0

11 STO Eligible := 0

12 ADR -103 address of Age

14 INN READ(Age)

15 ADR address of Age

17 VAL value of Age

18 LIT 0 push 0

20 GTR compare

21 BZE 74 WHILE Age > 0 DO

23 ADR -103 address of Age

25 VAL value of Age

26 LIT 18 push VotingAge

28 GTR compare

29 BZE 69 IF Age > VotingAge THEN

31 ADR -2 address of Voters[0]

33 ADR -1 address of Eligible

35 VAL value of Eligible

36 LIT 101 array size 101

38 IND address of Voters[Eligible]

39 ADR -103 address of Age

41 VAL value of Age

42 STO Voters[Eligible] := Age

43 ADR -1 address of Eligible

45 ADR -1 address of Eligible

47 VAL value of Eligible

48 LIT 1 push 1

50 ADD value of Eligible + 1

51 STO Eligible := Eligible + 1

52 ADR -104 address of Total

54 ADR -104 address of Total

56 VAL value of Total

57 ADR -2 address of Voters[0]

59 ADR -1 address of Eligible

61 VAL value of Eligible

62 LIT 1 push 1

64 SUB value of Eligible - 1

65 LIT 101 array size 101

67 IND address of Voters[Eligible-1]

68 VAL value of Voters[Eligible - 1]

69 ADD value of Total + Voters[Eligible - 1]

70 STO Total := Total + Voters[Eligible - 1]

71 ADR -103 address of Age

73 INN READ(Age)

74 BRN 15 to start of WHILE loop

76 ADR -1 address of Eligible

78 VAL value of Eligible

79 PRN WRITE(Eligible,

80 PRS ’ voters. Average age = ’

82 ADR -104 address of Total

84 VAL value of Total

85 ADR -1 address of Eligible

87 VAL value of Eligible

88 DVD value of Total / Eligible

89 PRN WRITE(Total / Eligible)

90 NLN output new line

91 HLT END.

Так же в фазу Back End входит простая оптимизация кода — удаление избыточного кода.

4.5 Процедуры и функции

4.6 Параллельное программирование