Министерство образования и науки РФ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Томский государственный университет систем управления и
радиоэлектроники» (ТУСУР)

Методические указания по выполнению практических и самостоятельных работ по дисциплине «Современные проблемы информатики и ВТ»

Для специальностей:
230100 - «Информатика и вычислительная техника»

Факультет: экономический
Профилирующая кафедра: экономической математики, информатики и статистики
Обеспечивающая кафедра: экономической математики, информатики и статистики

Разработчик:
доц. каф. ЭМИС Стась А.Н.

2012 г.

Содержание

Лексический анализ

Основная задача лексического анализа - разбить входной текст, состоящий из последовательности одиночных символов, на последовательность слов, или лексем, т.е. выделить эти слова из непрерывной последовательности символов. Лексический анализатор иногда называют сканером. Все символы входной последовательности с этой точки зрения разделяются на символы, принадлежащие каким-либо лексемам, и символы, разделяющие лексемы (разделители). Обычно все лексемы делятся на классы.

Примерами лексем являются числа (целые, восьмеричные, шестнадцатиричные, действительные и т.д.), идентификаторы, комментарии, операнды. Отдельно выделяются ключевые слова и символы пунктуации (иногда их называют символы-ограничители). Как правило, ключевые слова - это некоторое конечное подмножество идентификаторов. Другие типы идентификаторов – имена переменных. В некоторых языках (например, ПЛ/1) смысл лексемы может зависеть от ее контекста и невозможно провести лексический анализ в отрыве от синтаксического.

С точки зрения дальнейших фаз анализа лексический анализатор выдает информацию двух сортов: о последовательности лексем и о конкретных значениях отдельных лексем (идентификаторов, констант и т.д.). Как приавило строятся таблицы объектов – идентификаторов, констант, ключевых слов и т.д . Комментарии выбрасываются.

Работа лексического анализатора описывается с помощью детерминированных конечных автоматов. Однако непосредственное описание конечного автомата не всегда удобно, поэтому часто применяют регулярные выражения.

Опишем лексику простейшего языка программирования.

Все переменны целочисленны, нет описания типов. Константы также целочисленны. Идентификаторы начинаются с латинской буквы, далее латинские буквы или цифры. Но есть одномерные массивы.

; разделителиоператоров

Присваивание :=

Операции бинарные: +, -, *, / (div), \ (mod),<,>,<=,>=,!=,= (рез – 0,1)

A[i] – доступ к элементу массива – индексация (начальный индекс – 0).

Комментарии: |*…….*|

Команды:

INPUT <перем> - ввод

OUTPUT <выр> - вывод

IF <выр> THEN <оператор 1> [ELSE <оператор 2>]

WHILE <выр> DO <оператор>;

{составной оператор} (в паскале BEGIN…END)

FUNC<назв>(<параметр>) – начало функции

RET возвр. Знач. – возврат из функции
<назв. Функ.>(<парам>) – вызов функции внутри выражения.
HALT – прекращение выполнения программы.

Пример 1. Вычисление суммы от 1 до N

```
IINPUT n;
s:=0;
i:=1;
WHILE i<n DO {
   s:=s+i;
   i:=i+1
};
OUTPUT s
HALT;
```

Пример 2. Вычисление факториала.

```
INPUT N;
OUTPUT fact(n);
HALT;
FUNC fact(k);
IF k=1 THEN RET 1 ELSE RET(k*fact(k-1));
```

Описание лексики с помощью конечного автомата

	a..z, A..Z	0..9	;	п	+,-,/,\,	(	)	[	]	:	=	>,<	!	|	*	{	}
S	A	B	S	S	S	S	S	S	S	C	S	D	E	F	S	S	S
A	A	A	S	I	S	S (ф-ция)	S	S	S	C	S	D	E	F	S	S	S
B	Ош	B	S	S	S	Ош	S	Ош	S	Ош	S	D	E	F	S	Ош	S
C	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	S (:=)	Ош	Ош	F	Ош	Ош	Ош
D	A (<,>)	B (<,>)	Ош	S (<, >)	Ош	Ош	Ош	Ош	Ош	Ош	S (<=, >=)	Ош	Ош	F	Ош	Ош	Ош
E	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	S (!=)	Ош	Ош	F	Ош	Ош	Ош
F	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош	G	Ош	Ош
G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	H	G	G
H	G	G	G	G	G	G	G	G	G	G	G	G	G	Ком ком	G	G	G
I	A (нов)	B (нов)	S	I	S	S (ф-ция)	S	S	S	C	S	D	E	F	S	S	S

S- выделение лексемы

A – выделение идентификатора или ключевого слова.
B – выделение числовой константы
C – Возможно присваивание
D – простые или составные операции сравнения <,>,<=,>=
E – возможно операция “!=”.
F- возможно комментарий
G – комментарий
H – возможен конец комментария
I – возможен вызов функции

Таблица ключевых слов:
K1 - INPUT
K2 - OUTPUT
K3 - IF
K4 – THEN
K5 - ELSE
K6 - WHILE
K7 - DO
K8 - FUNC
K9 - RET
K10 - HALT.

Программирование лексических анализаторов

Лексический анализатор, как правило, вызывается как подпрограмма.

Тело ЛА представляет собой диаграмму переходов соответствующего конечного автомата. Отдельная проблема - анализ ключевых слов. Как правило, ключевые слова - это выделенные идентификаторы. Поэтому возможны два основных способа выделения ключевых слов: либо очередная лексема сначала диагностируется на совпадение с каким-либо ключевым словом и в случае неуспеха делается попытка выделить лексему из какого-либо класса, либо, наоборот, после выборки лексемы идентификатора требуется заглянуть в таблицу ключевых слов на предмет сравнения.

В некоторых языках (например, ПЛ/1 или Фортран) ключевые слова Обратим внимание на некоторые моменты работы с нашим языком. По окончании выделения идентификатора, проверяем имя в таблице ключевых слов. Если идентификатор заканчивается “(“ – имя функции между возможны пробелы, поэтому при подозрении на данный случай необходима задержка выделения идентификатора (нетерминал I)). Ключевые слова заменяем их кодами. Переменные и константы вносим в таблицы и заменяем ссылками на элементы таблицы. Сложные операнды (больше 1-го символа заменяем на одиночные символы).

Обратим внимание, что по окончании комментария мы должны обеспечить возврат в точку, предшествующую началу комментария. При нахождении ув состоянии «возможен комментарий» прекращается генерация символов 1-го внутреннего представления.

Примеры 1-х внутренних представлений.

1-е внутреннее представление, генерируемое лексическим анализатором имеет, как правило, свои особенности. Так, ключевые слова, имена переменных, константы заменяются ссылками на элементы соответствующих таблиц, а сложные операнды заменяются однозначными кодовыми символами.

Также необходимо сохранять значения констант для данной программы, например в конце описания после спецсимвола $. При реализации обычно проще выгрузить таблицу констант в отдельный файл.

Например, := - @, <= - ~, >= $, != - ^

Пример 1. Вычисление суммы от 1 до N

INPUT n;
s:=0;
i:=1;
WHILE i<=n DO {
  s:=s+i;
  i:=i+1
};
OUTPUT s
HALT;
	K1I1;
	I2@C1;
	I3@C2;
	K6I3~I1K7{
	I2@I2+I3;
	I3@I3+C3
	};
	K2I2;
	K10;
	$C1=0$C2=1$C3=1

Пример 2. Вычисление факториала.

INPUT n; OUTPUT fact(n); HALT; FUNC fact(k); IF k=1 THEN RET 1 ELSE RET k*fact(k-1);	K1I1; K2F1(I1); K10; K8F1(I2); K3I2=C1K4K9C2K5 K9I2*F1(I2-C3); $C1=1$C2=1$C3=1

Синтаксический анализ.

Задачи синтаксического анализа.

Основная задача синтаксического анализа - разбор структуры программы. Результатом будет представление на втором внутреннем языке, не зависящее (в отличие от представления на первом внутреннем языке) от синтаксиса программы. В процессе синтаксического анализа также обнаруживаются ошибки, связанные со структурой программы.
Синтаксис языков программирования, чаще всего описывается контекстно-свободной грамматикой, хотя бывают и исключения (shell – автоматный синтаксис, ALGOL-60 – КЗ-синтаксис). Очевидно, что задача является расширением задачи анализа КС-языка.

Описание синтаксиса языка

Опишем синтаксис нашего языка:

<программа>:-{<оператор>;}*[оператор]
<оператор>:-<составной оператор>|<простой оператор>
<составной оператор>:-{<программа>}
<простой оператор>:-<присваивание>| <ввод>| <вывод>| <условный оператор>| <цикл с предусловием> |<заголовок функции>| <возврат из функции>| <останов>
<присваивание>:-<переменная>:=<выражение>
<ввод>:-INPUT <переменная>
<вывод>:-OUTPUT <выражение>
<условный оператор>:-IF <выражение> THEN <оператор> [ELSE <оператор>]
<цикл с предусловием>:-WHILE <выражение> DO <оператор>
<заголовок функции>:-FUNC <имя функции> (<имя переменной>)
<возврат из функции>:-RET <выражение>
<останов>:-HALT
<выражение>:-<операнд>| (<выражение>) |
<выражение><операция><выражение>
<операнд>:-<переменная>| <константа>| <вызов функции>
<операция>:- +| -| *| /| \| <| >| <=|>=| =| !=
<переменная>:- <имя переменной>| <имя переменной>[<выражение>]
<вызов функции>:- <имя функции>(<выражение>)

Замеч. Лишние пробелы, комментарии, переводы строки и т.д. удалены на этапе лексического анализа.

Описание синтаксиса в виде КС-грамматике.
От описания в форме Б-Н легок перйти к описанию в виде КС-грамматики, с учетом представления на первом внутреннем языке:
В нашем случае имеем:

S-> A;S| A| л A -> B| C B -> {S} C –> D | E | F |G | H| I| J| K D -> L@M E -> k1L F -> k2M G ->k3Mk4A|k3Mk4Ak5A H -> k6Mk7A I -> k8N(O) J -> k9M K -> k10 L -> O| O[M] M -> P | (M)| MRM N -> fT O -> iT P -> L| U| V R -> +| -| *| /| \\| <| >| ~| $| =| ^ T -> 0T| 1T |…..| 9T| 0| 1|…| 9 U -> cT V -> N(M)	S – программа (последовательность операторов) A – оператор B – составной оператор C – простой оператор D – присваивание E – ввод F – вывод G – условный оператор H – цикл с предусловием I – заголовок функции J – возврат из функции K – останов L – переменная M – выражение N – имя функции O – имя переменной P – операнд R – операция T – номер (константы, переменной, ключевого слова или функции) U – константа V – вызов функции

Замечание. Используем малые буквы для обозначения ключевых слов, констант, имен переменных и функций в схематическом описании, т.к. они являются терминальными символами.

Простейшие приемы оптимизации грамматики

Простейшим способом оптимизации является уменьшение числа нетерминалов за счет упразднения тривиальных правил типа A -> B с

удалением недостижимых и бесполезных (для их выявления нужен более глубокий анализ) нетерминалов.

Проведем преобразования для нашей грамматики:

S-> A;S| A| л A -> B| C B -> {S} C –> D | E | F |G | H| I| J| K D -> L@M E -> k1L F -> k2M G ->k3Mk4A|k3Mk4Ak5A H -> k6Mk7A I -> k8N(O) J -> k9M K -> k10 L -> O| O[M] M -> P | (M)| MRM N -> fT O -> iT P -> L| U| V R -> +| -| *| /| \\| <| >| ~| $| =| ^ T -> 0T| 1T |…..| 9T| 0| 1|…| 9 U -> cT V -> N(M)	S -> A;S| A| л A -> {S}| L@M| k1L| k2M| k3Mk4A| k3Mk4Ak5A| k6Mk7A| k8N(O)| k9M| k10 L -> O| O[M] M -> L | U | V| (M)| MRM N -> fT O -> iT R -> +| -| *| /| \\| <| >| ~| $ =| ^ T -> oT| 1T|….| 9T| 0| 1|…| 9 U -> cT V -> N (M)
	S -> A;S| A| л A -> {S}| L@M| k1L| k2M| k3Mk4A| k3Mk4Ak5A| k6Mk7A| k8N(iT)| k9M| k10 L -> iT| iT[M] M -> iT | iT[M]| cT| N(M)| (M)| MRM N -> fT R -> +| -| *| /| \\| <| >| ~| $ =| ^ T -> oT| 1T|….| 9T| 0| 1|…| 9

Таким образом, мы получаем формальную грамматику, в соответствие с которой будем анализировать синтаксис. Для простоты сначала рассмотрим относительно простую задачу анализа текста программы на наличие синтаксических ошибок, которая как легко видеть сводится к обычному КС-анализу. Общие подходы к решению этой задачи были рассмотрены в рамках дисциплины ТА. Этот подход предполагает моделирование построение ие и моделирование МА. Здесь, мы подробнее рассмотрим применяемые на практике частные методике, как мы увидим частные случаи применения МА, позволяющие достичь линейно трудоемкости анализа с помощью детерминированного алгоритма (алгоритм МА недетерминированный).

Анализ текста программы по КС-грамматике

Для анализа синтаксиса языка могу быть использованы конечный или магазинный автоматы.
Левый и правый проходы определяются прямым или обратным просмотром цепочки. На практике чаще применяют левые проходы.

Левое порождение генерируется, если в каждый момент раскрывается самый левый нетерминал, правое порождение соответственно моделируется, если в каждый момент раскрывается самый правый нетерминал.
На примерах легко увидеть что нисходящий анализ генерирует левое порождение, а восходящий – правое.
В соответствие с типами проходов и порождений различают типы алгоритмов: LL, LR, RL, RR.
Часто, левые и правые порождения невозможно определить однозначно, в таких случаях говорят, что грамматика неоднозначна. На практике неоднозначность обычно приводит к неопределенности в порядке выполнения арифметических операций. На практике, для восстановления правильного порядка применяются дополнительные методы, т.е. непосредственно грамматика помогает только проверить отсутствие синтаксических ошибок в выражениях. Описания операторов однозначны.

Построение LL-анализатора по грамматике.

Идея детерминированного LL-анализатора: цепочка просматривается слева направо (от начала программы к концу), моделируется левое порождение, т.е. применяется нисходящий анализ. Детерминированность алгоритма пытаемся обеспечить на основании текущего символа входной цепочки, и возможно нескольких символов за текущим. Обратим внимание, что неоднозначными могут быть только операции замены нетерминала в стеке.
Рассмотрим подробнее идею на примере:

S -> S+S| S*S| (S)|a
a+(a*a)+(a*a)+a

Глядя на эту грамматику, очень трудно предположить, как сделать первый переход в зависимости от символа входной цепочки (к-й очевидно может быть только символом “a”). Главной причиной неудобства является левая рекурсия, т.е. возможность вывода нетерминала из самого себя без сдвига по входной цепочке за конечное число применения Л-переходов.

Примеры левой рекурсии: S->S+S S->S*S (за 1 переход)

S → Aa A → Bb | f B → Cc C → Dd | e D → Az
Здесь леворекурсивный цикл, вовлекающий A, B, C, D.
Чтобы заменить этот цикл прямой левой рекурсией, упорядочим нетерминалы следующим образом: S, A, B, C, D.
Рассмотрим все порождающие правила вида

Xi → Xj γ,

где Xi и Xj – нетерминалы, а Фи – смешанная цепочка. В отношении правил, для которых j ≥ i, никакие действия не производятся. Однако это неравенство не может выдерживаться для всех правил, если есть левый рекурсивный цикл. При выбранном нами порядке мы имеем дело с единственным нарушением в правиле:

D → Az

так как A предшествует D в этом упорядочении. Теперь начнем замещать A, пользуясь всеми правилами, имеющими A в левой части. В результате получаем

D → Bbz | fz

Поскольку B предшествует D в упорядочении, процесс повторяется:

D → Ccbz | fz

Затем он повторяется еще раз:

D → Ddcbz| ecbz | fz

Теперь грамматика выглядит так:

S → Aa
A → Bb |f
B → Cc
C→ Dd| e
D → Ddcbz |ecBz | fz

Далее, рассмотрим правила для D. Очевидно, что порождаемая из D цепочка будет начинаться на ecBz или fz, а далее от 0 до бесконечности раз будет повторятся подцепочка dcbz. Эту закономерность можно описать и так:

D -> ecBzE|fzE|ecBz|fz
E -> dcbzE|dcbz

Таким образом, мы избавились от левой рекурсии за счет добавления одного нетерминала E в грамматику. Действуя подобным образом, мы можем любую КС-грамматику избавить от левой рекурсии.

Пример. Избавим от левой рекурсии и преобразуем к удобному виду грамматику:

S->S+S|S*S|(S)|a

S->(S)+S|(S)*S|a+S|a*S|(S)|a

S->A+S|A*S|A
A->(S)|a

S->AB
A->(S)|a
B->+S|*S|л

S->AB
A->(S)|a

B->+AB|*AB|л

S->(S)B|aB
A->(S)|a
B->+AB|*AB|л

Теперь, мы можем составить таблицу рекомендуемых применений правил в зависимости от входной цепочки.

	+	*	(	)	a	Исч.
S			(S)B			aB
A			(S)			a
B	+AB	*AB	л	л	л	л

Таким образом, мы в каждый момент по одному символу входной цепочки можем принять однозначное решение о применяемом переходе.

Очевидно, что для того, что строить таблицу LL(1)-анализатора удобно по грамматике, где правила приведены к нормальной форме Грейбаха

В этом случае, LL(1)-анализатор можно построить, если первые терминалы

Метод рекурсивного спуска основан на моделировании LL(k) – анализа без необходимости полного моделирования МА. Такая возможность появляется за счет использования рекурсивных алгоритмов, в этом случае МА на самом деле моделируется неявно за счет размещения параметров вызовов в стеке (см. курс «программирование» - рекурсия).
Идея метода. Для каждого нетерминала создается функция, моделирующая анализ начиная с данного нетерминала. В случае если при раскрытии нетерминала встречается другой нетерминал происходит вызов соответствующей функции. Процесс продолжается до исчерпания входной цепочки (программы) или обнаружения ошибки. Необходимым условием реализации идеи – отсутствие левой рекурсии в грамматике (в этом случае анализ будет циклить). Анализ начинается с нетерминала S;
Пример (наш язык).

S -> A;S| A| л A -> {S}| L@M| k1L| k2M| k3Mk4A| k3Mk4Ak5A| k6Mk7A| k8N(iT)| k9M| k10 L -> iT| iT[M] M -> iT | iT[M]| cT| N(M)| (M)| MRM N -> fT R -> +| -| *| /| \\| <| >| ~| $ =| ^ T -> oT| 1T|….| 9T| 0| 1|…| 9	S -> A;S| A| л A -> {S}| L@M| k1L| k2M| k3Mk4A| k3Mk4Ak5A| k6Mk7A| k8N(iT)| k9M| k10 L -> iT| iT[M] M -> LZ|cTZ| N(M)Z| (M)Z| MZ N -> fT R -> +| -| *| /| \\| <| >| ~| $ =| ^ T -> oT| 1T|….| 9T| 0| 1|…| 9 Z->RM|л

Замечание. На самом деле в данном языке существует леворекурсивное правило M->MRM. Однако мы пока не будем обращать на него внимание по причнам, к-е рассмотрим в пункте 3.3.6.

```
Procedure SAnal; //анализатор S, т.е. всей грамматики
Begin
  If str<>'' then begin  //str – анализируемая строка    S->л (пустая строка –ОК)
  AAnal;
  If str – не исчерпана then begin  //если исчерпана – ОК      S->A
    If str[cur]<>';' then begin //cur – текущая позиция – глобальная переменная
       Res:=false; //результат анализа – глобальная переменная
       Halt;
    End;
    Inc(cur); //сдвигаем в строке в соответствии с ";"
    SAnal
  End
End;

Procedure Anal;
Begin
  Case str[cur] of
   '{': begin
     inc(cur);
     SAnal;
    If str[cur]<>'}' then begin
      res:=false;
      halt
    else  inc(cur)
   end;
   'I': begin
      LAnal;
      If str[cur]<>'@' then begin
       res:=false;
       halt
      else  inc(cur);
      MAnal;
    end;
    'k':  begin
       case str[cur+1] of //смотрим один символ вперед
         '1': begin
       ……..//здесь надо смотреть еще 3-й символ – k1 или k10
         end;
        ……
       end
    end
```

```
……..
  end
end;
```

Проверка выражений.

Отдельно рассмотрения заслуживает задача рассмотрения выражений. На это есть несколько причин – выражения состоящие из операций, операндов и скобок присутствуют в большинстве языков программирования, кроме того, как уже отмечалась выше грамматика, описывающая выражения неоднозначно, что объясняется наличием приоритетов операций, которые грамматика не учитывает.

Простой алгоритм Дейкстры позволяет преобразовать инфиксную запись в постфиксную, иначе называемую ОПЗ (обратная польская запись), ОПС, ОПН, РПЗ, РПС, РПН.

1. Проверяется очередной символ во входной строке.
2. Если это операнд, то он передается в выходную строку.
3. Если это открывающая скобка, то она заносится в стек с приоритетом нуль.
4. Если это операция, то ее приоритет сравнивается с приоритетом операции, находящейся на вершине стека. Если приоритет новой операции больше, то эта новая операция заносится в стек. В противном случае берется операция с вершины стека и помещается в выходную строку, после этого повторяется сравнение с новыми верхними элементами стека до тех пор, пока на вершине стека не окажется операция с приоритетом, меньшим, чем у текущей операции, или пока стек не станет пустым. После этого текущая операция заносится в стек.
5. Если текущий символ во входной строке является закрывающей скобкой, то операции из стека последовательно переносятся в выходную строку до тех пор, пока на вершине стека не появится открывающая скобка; эта открывающая скобка отбрасывается. Если стек опустеет, а открывающая скобка не встретилась – в выражении ошибка.
6. Если выражение закончилось, то из стека последовательно переносятся в выходную строку все оставшиеся в нем операции. Если при этом, в стеке встретится открывающиеся скобка, то в выражении допущена ошибка.

Очевидно, что контроль за скобочной структурой в алгоритме производится автоматически. Но для полной уверенности в корректности выражения следует контролировать возможных нарушений корректности инфиксной записи. Для этого следует убедиться в отсутствии следующих ошибок:

1) Выражение не начинается с операции (если все операции бинарные).
2) Выражение не заканчивается операцией.

3) В строке нет 2 операций подряд или разделенных исключительно скобками
4) В строке нет 2 операндов подряд или разделенных исключительно скобками.
5) После открывающей скобки не следует операция (если все операции бинарные).
6) После операции нет закрывающей скобки.
7) После открывающей скобки нет закрывающей скобки
8) После закрывающей скобки нет открывающей скобки
9) Корректность своих операндов

Очевидно, что для контроля следует отдельно проверить начальный и заключительный символы в строке, а по ходу просмотра строки, содержащей выражения сохранять информацию являлся ли предыдущий символ открывающей скобкой, являлся ли он закрывающей скобкой, а также был ли, последний нескобочный символ операцией или операндом.

Замеч. Операнды мы называем символами условно, на самом деле их запись состоит из нескольких символов, начинающихся с “I”, “C” или “F” (смотри «лексический анализ».

Такой алгоритм легко реализовать на этапе рекурсивного спуска, так как метод допускает разные подходы к раскрытию разных нетерминалов. В случае нашего языка, используем алгоритм Дейкстры для анализа нетерминала M – выражение.

Можем установить следующие приоритеты:

- % - 4
- * , / , \ - 3
- +, - -2
- <,>,<=,>=,=,!= - 1

2-е внутренне представление и его генерация в процессе СА

Известно, что кроме проверки программы на наличие синтаксических ошибок, на этапе СА должны быть решена задача генерации текста программы на втором внутреннем языке, который является синтаксически независимым относительно синтаксиса исходного текста, и в тоже время остается аппаратно-независимым. На самом деле, существует несколько подходов ко второму внутреннему представлению.

Требования к нему формулируется исходя, с одной стороны, из возможности эффективной генерации этого представления на этапе СА, с другой стороны с точки зрения возможности эффективной интерпретации программы на втором внутреннем языке, возможности эффективной генерации команд. Учитывается и возможность эффективной организации контекстного анализа, и возможность эффективной аппаратно-независимой оптимизации.

Классическим является представление на основе расширения ОПЗ. В настоящее время, распространение получают байт-коды, например вирутуальная машина Java.

В 3.3.5 рассмотрено применение ОПЗ для представления арифметических выражений. При этом введена команда вызова функции %. Для того, чтобы в последствие, информация о вызове не была потеряна необходимо вместо номера функции, задать адрес перехода, для чего вводится понятие метки.
Замеч. Иногда, есть необходимость различать вызов с возвращаемым результатом и без него – в этом случае вводятся разные команды.
Очевидно, что роль метки здесь аналогично языку Pascal. Концепцию меток можно применить, и для представления условных операторов и циклов. Для этого вводим две команды,
J – jmp – безуловный переход (формат aj, a –метка, команда унарная).
Z – jz – условный переход (abz, a – параметр перехода, b – метка).
В первом случаем переход по метке осуществляется в любом случае, во втором случае, только если a=0.
Тогда условную конструкцию можно представить следующим образом:
If <выражение> then <оператор>;
//запись в ОПЗ
If <выражение> then <оператор> else <оператор>;
//запись в ОПЗ
А цикл с предусловием следующим образом:
While <выражение> do <оператор>
//запись в ОПЗ
Легко видеть, что и другие формы циклических операторов можно также организовать с помощью аналогичного подхода.
Аналогично, в ОПЗ можно ввести специальные операции для реализации специфических команд. Так, например, для нашего языка придется ввести:
@ - присваивание (ab@ - a:=b)
I – INPUT (ai – ввод значения a - унарная)
O – OUTPUT (ao – вывод a - унарная).
H – HALT (h – 0-арная).
R – RET (ar – унарная).
G – Get Param – получение значения параметра функции (ag – унарная)
Что касается генерации ОПЗ на этапе СА, то соответствующий спосб для арифметических выражений рассмотрен в 3.3.5. Для остальных конструкций дело обстоит проще. Так, при рекурсивном спуске вывод необходимых данных в ОПЗ легко встраивается в процедуру анализа того или иного нетерминала (подробнее на практике). В алгоритмах, основанных на таблицах раскрытия (нисходящий анализ) или свертки (восходящий анализ) запись соответствующих конструкций в ОПЗ легко осуществить в момент соответствующей замены в стеке.
Пример ОПЗ, соответствующей программе (сумма нат. чисел от 1 до n)

INPUT n;	K1I1;	I1OI

s:=0;
i:=1;
WHILE i<=n DO {
  s:=s+i;
  i:=i+1
};
OUTPUT s
HALT;
	I2@C1;
	I3@C2;
	K6I3~I1K7{
	I2@I2+I3;
	I3@I3+C3
	};
	K2I2;
	K10;
	$C1=0$C2=1$C3=1
		I2C1O@
		I3C2O@
		I3I1O~L2OZ (метка L1 – перед началом)
		I2I2I3O+O@
		I3I3C3O+O@
		L1OJ
		I2OO (метка L2 – перед началом)
		OH

Замеч 1.. Реально, обозначения меток задаются в таблице меток, которые выгружаются в отдельный файл.

Замеч. 2 O – для удобства дальнейшей интерпретации ставим перед каждой операцией.

Замеч. 3. 2-е внутреннее представление рекомендуется сохранять в файле, структура которого близка к бинарной, конкретно номера меток, переменных, констант и функций будем хранить в виде символов с соответствующими номерам кодами.

Пример ОПЗ, соответствующей программе (вычисление факториала)

INPUT n; OUTPUT fact(n); HALT; FUNC fact(k); IF k=1 THEN RET 1 ELSE RET k*fact(k-1);	K1I1; K2F1(I1); K10; K8F1(I2); K3I2=C1K4K9C2 K5K9I2*F1(I2-C3); $C1=1$C2=1$C3=1	I1OIk F1I1O%OO OH I2OG I2C1O=L1OZC2ORL2OJ I2F1I2C3O-O%O*OR (Метка L1 – перед началом) (метка L2 – в конце)

Замеч. Метка вызова функции прописывается в момент распознания соответствующего заголовка, поэтому метки функций обозначаем особо.

Интерпретация ОПЗ.

В случае, если речь идет об интерпретировании, то применяют алгоритм интерпретации программы для соответствующего второго внутреннего представления. В качестве примера рассмотрем задачу интерпретации ОПЗ.

Алгоритм интерпретации арифметических выражений в постфиксной форме достаточно прост. Строка анализируется слева направо. Если текущий символ – операнд, то он помещается в стек, если текущий символ – операция, то из стека извлекается количество операндов в соответствие с арностью операции, после чего операция выполняется и результат помещается в стек. По окончанию анализа в стеке будет результат выполнения выражения.

Пример.

(5+3)*7-2*6/3+2*4

5 3 + 7 * 2 6 * 3 / - 2 4 * +

Содержимое стека: 5 3 8 7 56 2 6 56 12 3 56 4 52 2 4 52 8 60

Легко видеть, что алгоритм обладает линейно относительно длины цепочки трудоемкостью. Именно, наличие такого алгоритма и делает ОПЗ универсальной формой представления алгоритмических конструкций, но задача состоит, чтобы приспособить данный подход и к расширенному понятию ОПЗ.

Очевидно, что операцию присваивания легко вписать в приведенный выше алгоритм, если будет разработана логика выполнения данной операции. Для этого, вводим таблицы значений констант различных типов (инициализируются из файла значений констант, полученных на этапе ЛА), а также таблицы значение идентификаторов различных типов, точнее речь идет о выделяемых для хранения констант и переменных областях памяти. Информация о типах должна быть установлена на этапе КА и является значимой, так как именно типом определяется объем памяти, требуемый для хранения значения переменной, а также способы интерпретации хранимых значений.

В этом случае, выполнении е присваивания фактически означает копирование в соответствующую область памяти значение выражения. При этом никакого результата в стек не вносится

Пример.

A:=3+4; A 3 4 + @

Содержимое стека: A 3 4 A 7 пусто

В результате выполнения последней операции по соответствующему адресу в соответствующей области памяти будет помещено значение 7.

4.2.3. Ввод, вывод, останов.

Данные команды легко вписываются в алгоритм интерпретации. При исполнении ввода (I) из стека извлекаем адрес ввода. Далее запрашиваем значении и сохраняем по адресу.

Для команды вывода (O) извлекаем значение из стека (точнее в стеке будет находится адрес, по котрому ищем значение, см. далее) и печатаем на стандартный вызов.

Команда останова (H) приводит к окончанию интерпретации.

4.2.4. Хранение информации. Индексация и расчет адресов.

Для корректной интерпретации в стек должны помещаться не абстрактные идентификаторы, а адрес в памяти (указатель) по которому хранится значение. Данная информация для переменных извлекается из соответствующих таблиц. Вместо промежуточных значений в стек также помещаются адреса, по которым они хранятся. Это касается и арифметических выражений. Для хранения промежуточных результатов соответственно выделяется специальная область памяти.

Пример:

(a+b)*c – d*e

При выполнении перации доступа к элементу массива по индексу (E) из стека извлекается адрес, по которому находится индекс элемента, а также базовый адрес. Результат операции – адрес требуемого элемента. Для расчет адреса используются классические правила адресной арифметики. Так, если размер элемента – 2 байта, а индексация начинается с позиции – ‘0’, то

<адрес элемента>=<базовый адрес>+2*<индекс элемента>

К переходам традиционно относят операции J, Z, %. Что касается первой операции, то она состоит в выталкивании из стека одного значения, которое является адресом, по которому хранится значение метки. Дале по таблице меток, сгенирированной на этапе СА, определяется точка перехода. Далее, анализ продолжается не со следующего символа в ОПЗ, а с символа, на который указывает метка, то есть просто изменяется порядок анализа. Для команды Z выталкивается 2 значения из стека – адрес значения метки и адрес управляющего значения. Если управляющее значение равно нулю, то аналогично команде J отрабатывается переход по метке, в противном случае интерпретация продолжается без нарушения команды.

Очевидно, что в этом случае нарушается линейность количества выполняемых операций в зависимости от длины ОПЗ. Трудоемкость будет определяться трудоемкостью алгоритма, реализованного в ОПЗ.

Пример.

WHILE i<=n DO {
  s:=s+i;
  i:=i+1
};
	I3I1O~L2OZ (метка M1 – перед началом)
	I2I2I3O+O@
	I3I3C3O+O@
	L1OJ

Операция % несколько сложнее. При вызове функции нужно помимо передачи управления по метке последовательно загрузить в стек вызовов (дополнительный стек) точку возврата, состояние программы и значение (я) параметра (ов). Под точкой возврата понимается адрес, по которому будет передано управление по команде ret, им является следующий элемент ОПЗ после операции %. Что понимать под состоянием программы зависит от конкретного языка и механизма хранения локальных переменных. Если в языке есть понятие области действия, то переменные с локальной областью действия (выявленной в КА) также хрянятся в стеке. В ассемблере вызов команды call подразумевает также сохранение в стеке значений системных регистров, чтобы управляющие команды внутри подпрограммы не влияли на управление основной программой. В рассматриваемом нами языке нет определений областей видимости, поэтому здесь каждая переменная может быть изменена внутри процедуры, и соответственно для обеспечения автономности подпрограммы придется сохранять значения всех переменных в стеке. По такому принципу работает, к примеру, интерпретатор скриптового языка php, однако там есть возможности исключить переменную из этого локального списка, применив описатель global. Значения параметров помещается в стек в последнюю очередь, что позволит их извлечь после передачи управления (команда G). Если параметров несколько, то важно четко соблюдать установленный порядок («справо налево» - C или «слева

направо» - Pascal). Команда позволяет извлечь очередной параметр, который был запомнен в стеке. В некоторых языках, например в Паскале существует два способа передачи параметров – прямой и ссылочный (косвенный – var).

Команда G позволяет получить очередное значение из стека. Соответственно исполнение команды состоит в извлечении значения из основного стека – адреса назначения (куда извлекаем), а из стека вызовов – значение параметра.

Команда R обеспечивает корректный возврат. Логика этой команды должна соответствовать команде %. Так, если в стеке вызовов были запомнены какие-то значения, то они выполняется восстановление этих значений из стека по исходным адресам. Если в стеке вызовов сохраняются локальные переменные, то соответствующая область очищается. Кроме того, из стека извлекается метка возврата для передачи управления. Если тип подпрограммы- функция (в pascal-терминологии), то после выполнения команды в основном стеке останется результат, являющийся параметром команды R.

На самом деле, можно обойтись одним стеком. Однако, если мы реализуем интерпретатор на типизированном языке, это не очень удобно, т.к. элементы основного стека – указатели, а элементы дополнительно стека – значения. В случае, если эти значения могут быть разных типов – реализация усложняется.

ОПЗ и машинно-независимая оптимизация

ОПЗ также легко позволяет провести машинно-независимую оптимизацию.

Алгоритм подобен тому, который применяется для вычисления выражений. Мы просматриваем выражение слева направо.

Пока есть символы для чтения:

- Читаем очередной символ.
- Если символ является адресом, по которому хранится определенное значение, помещаем это значение в стек.
- Если символ является адресом, по которому хранится неизвестное значение, считая что переменная имеет значение **nil**, помещаем символ в стек.
- Если символ является операцией:

1) (если все аргументы операции, лежащие в стеке, имеют значение, отличное от **nil** и это не операция I, O, H, J, Z, %, G, R) выталкиваем аргументы оператора из стека и помещаем в стек адрес результата операции;

2) если имеем операцию I, O, H, J, Z, %, G, R (если хотя бы один из аргументов имеет значение **nil**) считая что результат операции **nil**, кладём символ операции в стек.

После того, как всё выражение просмотрено, то, что осталось в стеке, является оптимизированным выражением (операторы выражения лежат в стеке в обратном порядке).

Пример. a+5*2

a 5 2 * +

a 5 2 a 10 a 10 +

Таким образом, применяя этот алгоритм после СА мы сможем упростить второе внутреннее представление, что является примером машинно-независимой оптимизации.

Генерация команд.

Особенности задачи.

Проблема генерации команд возникает на заключительных этапах компиляции. Задача очевидна – анализируя 2-е внутренне представление – сформировать исполняемый модуль для требуемой платформы.

Здесь, мы намеренно не будем останавливаться на структуре этих модулей для тех или иных платформах, эти особенности рассматриваются при обучении архитектуре компьютера и операционным системам. Наша задача – методология получения команд, которая предполагает генерацию 3-го внутреннего представления, близкого к реальной машинной программе (ОПЗ и виртуальная машина Java используют иной – стековый принцип). Иногда, компилятор минует стадию 2-го внутреннего представления и непосредственно в процессе СА генерирует 3-е внутреннее представление, однако этот подход затрудняет КА и снижает возможности машинно-независимой оптимизации.

В классическом варианте генерация команд состоит из двух предъэтапов – генерация 3-го внутреннего представления по оптимизированному второму внутреннему представлению, при возможности оптимизация кода на этапе 3-го внутреннего представления, а также сборка оптимально кода в соответствие с используемой платформой. Понятие платформы, напомним, инкапсулирует архитектуру вычислительной системы и системное ПО. Рассмотрим данные этапы.

Тетрадное и триадное представления и их получение.

Алгоритм интерпретации расширенной ОПЗ можно рассмотреть, как процесс выполнения определенных операций над определенным количеством аргументов (как правило, 0, 1 или 2 аргумента, в ином случае – операцию всегда можно логически разбить на несколько операций с двумя аргументами.).

Тогда, легко видеть, что схема вычисления представима в виде последовательности четверок

<Операция> <операнд 1><операнд 2><результат>

Замеч 1. Последовательность здесь может быть любой.

Замеч. 2. Если арность операции меньше 2, то один или два операнда будут пустыми, результат также может быть пустым

Замеч. 3. Вместо операндов и результата на практике прописываются адреса, по которым они хранятся.

Такие четверки называются тетрадами.

Пример. Сумма нат чисел от 1 до n.

I1OI	OI Ptr(I1) <> <>
I2C1O@	O@ Ptr(I2) Ptr(C1) <>
I3C2O@	O@ Ptr(I3) Ptr(C2) <>
I3I1O~L2OZ (метка L1 – перед началом)	O~ Ptr(I3) Ptr(I1) Ptr(P1) (метка L1 нна эту команду) OZ Ptr(P1) Ptr(L2) <>
I2I2I3O+O@	O+ Ptr(I2) Ptr(I3) Ptr(P2) O@ Ptr(I2) Ptr(P2) <>
I3I3C3O+O@	O+ Ptr(I3) Ptr(C3) Ptr(P3) O@ Ptr(I3) Ptr(P3) <>
L1OJ	OJ Ptr(L1) <> <>
I2OO (метка L2 – перед началом)	OO Ptr(I2) <> <> (метка L2 на эту команду)
OH	OH <> <> <>

Из примера видно, что при генерации последовательности тетрад по ОПЗ, необходимо обеспечить проставление меток для тетрад так, чтобы их содержание не изменилось (смещение в ОПЗ в тетрадном представлении разное). В качестве варианта содержанием метки здесь может быть номер теирады или ее смещение, которые пропорциональны друг другу, так как все тетрады имеют одинаковую длину.

Видно, что логика тетрад уже очень сильно напоминает логику машинных команд (программирование на ассемблере), особенно RISC-процессора. Однако, возникает вопрос, нет ли возможности оптимизировать код в тетрадном представлении. Действительно, вдруг есть заведомо избыточные команды. Здесь, на помощь приходит запись тетрардного представления в виде триад. Подход очень прост – отбрасывается поле «результат», а ссылки на промежуточные данные заменяются просто ссылками на номер триад, также реализуются и метки.

Очевидно, что между триадным и тетрадным представлениями существует взаимно-однозначное соответствие.

Пример

OI Ptr(I1) <> <>	1 OI Ptr(I1) <>
O@ Ptr(I2) Ptr(C1) <>	2 O@ Ptr(I2) Ptr(C1)
O@ Ptr(I3) Ptr(C2) <>	3 O@ Ptr(I3) Ptr(C2)
O~ Ptr(I3) Ptr(I1) Ptr(P1) (метка L1 на эту команду)	4 O~ Ptr(I3) Ptr(I1)
OZ Ptr(P1) Ptr(L2) <>	5 OZ T4 Ptr(L2)
O+ Ptr(I2) Ptr(I3) Ptr(P2)	6 O+ Ptr(I2) Ptr(I3)
O@ Ptr(I2) Ptr(P2) <>	7 O@ Ptr(I2) T6
O+ Ptr(I3) Ptr(C3) Ptr(P3)	8 O+ Ptr(I3) Ptr(C3)
O@ Ptr(I3) Ptr(P3) <>	9 O@ Ptr(I3) T8
OJ Ptr(L1) <> <>	10 OJ Ptr(L1) <>
OO Ptr(I2) <> <> (метка L2 на эту команду)	11 OO Ptr(I2) <>
OH <> <> <>	12 OH <> <>

	L1- 4 L2-11

В триадном представлении легко исключить лишние команды.

Прежде всего разделим команды на те, что производят действия и не производят. В нашем случае, к первой группе будут относится – I, O, H, Z, J, %, G, R, @. Ко второй группе все остальные. Очевидно, что триады команд второй группы, на которые нет ссылок в других триадах можно исключить. Затем повторять проверку до тех пор, пока не будут исчерпаны возможности исключения триад.

Существуют и иные приемы оптимизации на уровне триад, а следовательно и на уровне тетрад, например выявление практически недостижимых команд в силу расположения меток и инструкции останова, замены нескольких триад одной, если это возможно.

И так, если сформировано триадное или терадное представление, для окончания компиляции необходимо сформировать файл нужной структуры, состоящей из нужных команд, так называемый исполняемый модуль (например, exe или com).

Для примера вспомним структуру исполняемого модуля в Dos (из дисциплины «архитектура компьютера»). Модуль состоит из трех сегментов – код, данные и стек. Сегмент данных содержит ту информацию, которая на этапе 2-го и 3-го внутренних представлений хранится в таблицах значений – значения переменных, констант и т.д. Сегмент кода содержит непосредственно исполняемые команды. Сегмент стека – выделяемая область памяти, работающая, как накопитель стекового типа, что позволяет реализовать ряд операций, например вызовы.

И так, очевидно, что для того, чтобы сформировать сегмент данных достаточно информации обо всех данных, которая генерируется еще на этапе контекстного анализа. Под переменные просто выделяется память. Под константы происходит выделение памяти и инициализация значений.

Стековый сегмент – это просто выделенная область памяти.

Сегмент кода в наипростейшем случае генерируется так: каждой тетрад приводится в соответствие последовательность команд, определяемая архитектурой центрального процессора.

Пример

```
+ a b c

mov ax, a
add ax, b
mov c, ax
```

В более развитых алгоритмах, когда ставится задача генерации как можно более оптимального кода – среди тетрад ищутся фрагменты, соответствующие шаблону, для которого определены оптимальные коды.