Классическая теория компиляторов

Лекция 7

ИНТЕРПРЕТАТОРЫ

Интерпретатор – это программа, которая

- транслирует исходную программу на языке высокого уровня во внутреннее представление;
- выполняет (интерпретирует) программу, представленную на этом внутреннем языке.

Интерпретаторы хороши для обучения (когда большая часть времени уходит на отладку) и для тех языков, в которых много времени уходит на работу системных п/программ (работа с матрицами, базами данных и т.п.).

Достоинства интерпретаторов

- простота (не надо реализовывать генерацию объектного кода);
- удобство и простота отладки программ – все внутренние структуры нам доступны (прежде всего – это доступ к таблице символов в любой момент времени), легки трассировка, отслеживание обращений к меткам и переменным и т.п.;
- возможность включения интерпретируемых выражений в процессе выполнения программы (самомодификация программы).

=> Интерпретаторы наиболее часто применяются при разработке новых и сложных языков.

Недостатки интерпретаторов

- Малая, "по определению", скорость выполнения программы
- Для устранения этого недостатка приходится платить упрощением синтаксиса языка, т.е. его грамматики.

Аргументы

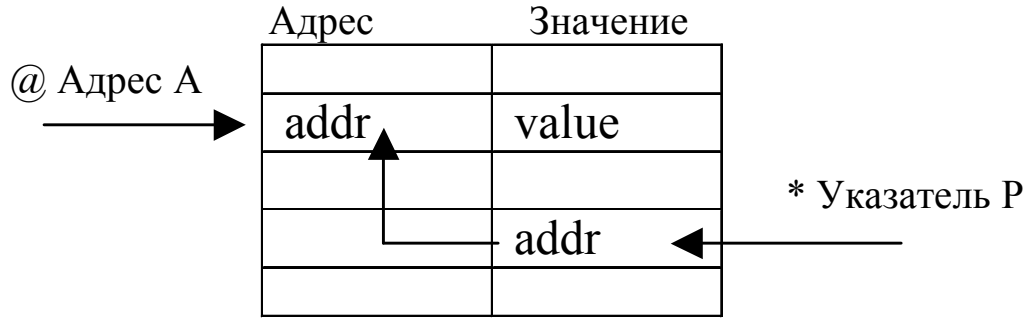
Наиболее эффективной формой внутреннего представления программы для интерпретатора является ПФ записи. Основной частью интерпретатора является переключатель CASE:

```
while TRUE do
begin
case gettype(P[n]) of
        operand: Push(S,P[n]);
        operator: arg1 := Pop(); arg2 := Pop(); Push(arg1 @ arg2);
        else: error();
    endcase
    n := n+1
end
```

Проблема представления в стеке аргументов различных типов данных.

4 типа аргументов:

- константы,
- имена,
- адреса переменнь
- указатели.



Хранение операндов

1. Различать типы данных, используя аналогию с байт-кодом. Либо хранить аргумент, предваряя его элементом, указывающим тип.
2. Тип элемента можно держать в таблице имен, и тогда в польской форме мы будем хранить лишь адреса.

Необходим механизм проверки типов и соответствующих функций конвертирования:

cvPV (pointer to value) – конвертация указателя в значение;

cvPA (pointer to address) – конвертация указателя в адрес;

cvAV (address to value) – конвертация адреса в значение.

Для увеличения эффективности выполнения программы можно заранее вставлять в ПФ процедуры конверсии типов.

Виды интерпретаторов

1. Интерпретатор, как самостоятельная программа, что зачастую создает некоторые проблемы, связанные с мобильностью программного обеспечения (один исполняемый файл удобнее пары – исходный текст программы плюс интерпретатор).
2. "Скрытые" или "неявные" интерпретаторы. Формируется исполняемый файл, содержащий в себе как непосредственно интерпретатор, так и исходный текст программы. Внешне мы имеем исполняемый модуль, который, занимается интерпретацией.
3. Виртуальные машины

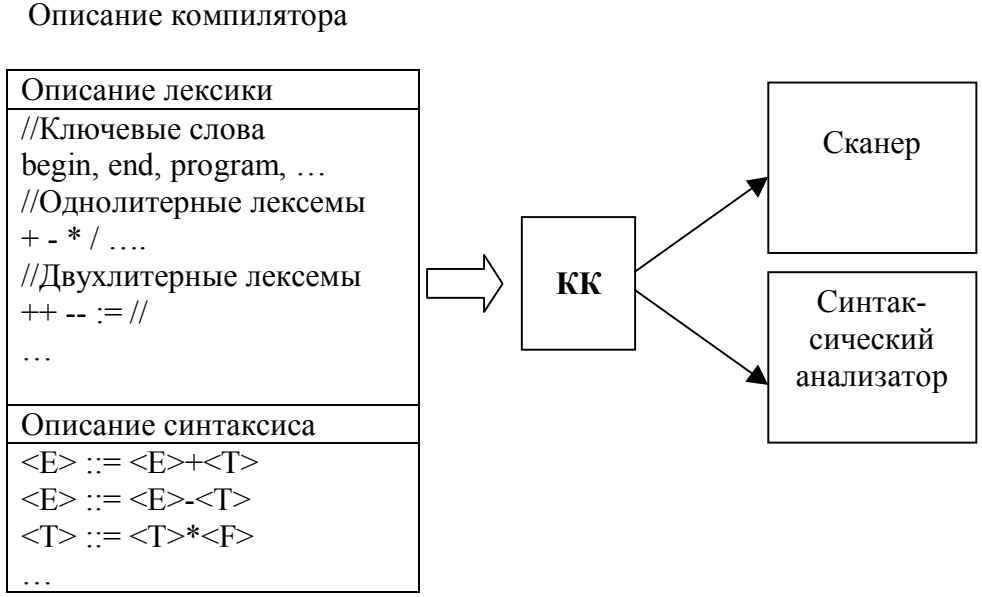
КОМПИЛЯТОРЫ КОМПИЛЯТОРОВ

КК – система, позволяющая создавать компиляторы. КК – это некий **инструментарий** программиста, помогающий создавать компиляторы. Автоматизация рутинных процедур.

- Имея регулярную грамматику лексической структуры можно, *автоматически* сгенерировать сканер в виде КА.
- Имея грамматику синтаксической структуры, можно написать *универсальную* процедуру синтаксически управляемого перевода (или создать *универсальный* МП-автомат).

Для этого потребуется некоторое описание двух грамматик – грамматики для сканера и грамматики для синтаксического анализатора.

Это описание можно хранить в некотором файле. Тогда получим специальный входной язык – язык описания компилятора.



Пример автомата

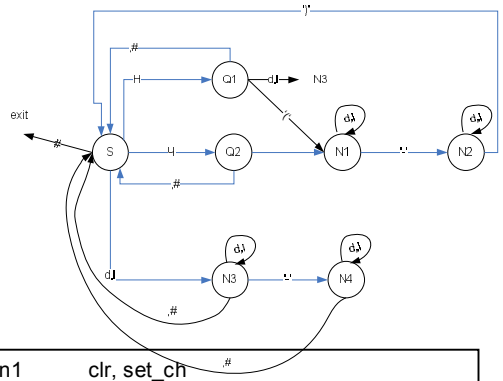
```
; Конфигурация распознающего автомата
; Входной алфавит
sD = "0123456789"
sL = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЫЭЬЪЮЯабвгдеёжзийклмнопрстуфхцчшщыэьъюя/_";
sN = "Н"
sC = "Ч"
sMinus = "-"
sLsk = "("
sRsk = ")"
sZpt = ","
sEol = "#"
sSpace = " "
sPoint = "."
#

; Состояния
s, q1, n1, n2, n3, n4
terminate, error
#

; Начальное состояние
S
#
; Терминальные состояния
terminate, error
#
; Переходы
S  sN   -> q1                          clr, dreset, acc
S  sC   -> q2                          clr, dreset, acc

S  sD   -> n3                          clr, dreset, acc
S  sL   -> n3                          clr, dreset, acc
S  sEol -> terminate
S  sZpt -> S
S  sSpace -> S

q1 sLsk -> n1          clr, set_nch
q1 sZpt -> S                           ADD
q1 sEol -> S          ADD, ungetch
q1 sZpt -> S          ADD, ungetch
q1 sD   -> n3          acc
q1 sL   -> n3         acc
```




```
q2 sLsk -> n1          clr, set_nch
q2 sZpt -> S                                ADD
q2 sEol -> S          ADD, ungetch
q2 sZpt -> S          ADD, ungetch
q2 sD   -> n3          acc
q2 sL   -> n3         acc

n1  sD      -> n1      acc
n1  sL      -> n1      acc
n1  sMinus   -> n2    set_n1, clr

n2  sD      -> n2      acc
n2  sL      -> n2      acc
n2  sRsk     -> S      set_n2, ADD

n3  sD      -> n3      acc
n3  sL      -> n3      acc
n3  sZpt     -> S      set_n1, set_n2, ADD
n3  sEol     -> S      set_n1, set_n2, ADD, ungetch
n3  sMinus   -> n4      set_n1

; Опасное место: игнорирование точки
n3  sPoint   -> n3

n4  sD      -> n4      acc
n4  sL      -> n4      acc
n4  sZpt     -> S      set_n2, ADD, ungetch
n4  sEol     -> S      set_n2, ADD, ungetch
#
```

Грамматика XQL

```
; ГРАММАТИКА XQL

<OPER> ::= <OPER> , <OPER>

;if C then P1 else P2 endif
;P1:
<PROGR> ::= <OPER>    ?'ELSE'     :push.L2.push.1.push.BR.do.L1
;P2:
<PROGR> ::= <OPER>    ?'ENDIF'    :do.L2

<PROGR> ::= <OPER>    ?','.<not>.'ELSE'.<not>.'ENDIF'.<not>.<and>.<and>
;-------------------------------------------------------------
<OPER> ::= STOP                    :push.0.push.EXIT
<OPER> ::= EXIT                    :push.0.push.EXIT
<OPER> ::= QUIT                    :push.0.push.EXIT
<OPER> ::= BR BRZ SICAFIL ELIST EXECUTE STARTSELECT STARTFROM
        STARTWHERE
;-------------------------------------------------------------
<OPER> ::= FOR <i> FROM <EXPR> TO <EXPR> DO <PROGR> ENDFOR
<OPER> ::= IF <IFCOND> THEN <PROGR> ELSE <PROGR> ENDIF
<IFCOND> ::= <EXPR>  ?'THEN'      :push.L1.push.2.push.BRZ
;-------------------------------------------------------------
<OPER> ::= WRITE <EXPR>    :push.1.push.WRITE
<OPER> ::= WRITELN <EXPR>  :push.1.push.WRITELN
<OPER> ::= READ  <EXPR>    :push.1.push.READ
<OPER> ::= SYSTEM <EXPR>   :push.1.push.SYSTEM

<OPER> ::= <i> = <EXPR>  :push.2.push.=

<ELIST> ::= <ELIST> , <ELIST>
<ELIST> ::= <EXPR>     ?','  :push.0.push.ELIST.push.0.push.SICAFIL
<ELIST> ::= <EXPR>           :push.0.push.ELIST
```

```
<OPER> ::= <STSELECT> <ELIST> <STSFROM> <ELIST> ?'<end>'
        :push.0.push.SELECT
<OPER> ::= <STSELECT> <ELIST> <STSFROM> <ELIST> <STSWHERE> <ELIST>
        :push.0.push.SELECT

<STSELECT> ::= SELECT   :push.0.push.STARTSELECT.push.0.push.SICAFIL
<STSFROM>  ::= FROM     :push.0.push.STARTFROM.push.0.push.SICAFIL
<STSWHERE> ::= WHERE    :push.0.push.STARTWHERE.push.0.push.SICAFIL

; АРИФМЕТИЧЕСКОЕ ВЫРАЖЕНИЕ

<EXPR> ::= <E>
        ?'LIKE'.<not>.'AND'.<not>.'OR'.<not>.'EQ'.<not>.'>'.<not>.'>='.<not>.'<='.<not>.'
        <>'.<not>.'<'.<not>.'+'.<not>.'-
        '.<not>.'*'.<not>.'/'.<not>.')'.<not>.<and>.<and>.<and>.<and>.<and>.<and>.<and
        >.<and>.<and>.<and>.<and>.<and>.<and>

;БИНАРНЫЕ ОПЕРАЦИИ НИЗКОГО ПРИОРИТЕТА

<E> ::= <E> <> <T>   ?'*'.<not>.'/'.<not>.<and>  :push.2.push.<>
<E> ::= <E> <= <T>   ?'*'.<not>.'/'.<not>.<and>  :push.2.push.<=
<E> ::= <E> >= <T>   ?'*'.<not>.'/'.<not>.<and>  :push.2.push.>=
<E> ::= <E> < <T>    ?'*'.<not>.'/'.<not>.<and>  :push.2.push.<
<E> ::= <E> > <T>    ?'*'.<not>.'/'.<not>.<and>  :push.2.push.>
<E> ::= <E> EQ  <T>  ?'*'.<not>.'/'.<not>.<and>  :push.2.push.EQ
<E> ::= <E> OR  <T>  ?'*'.<not>.'/'.<not>.<and>  :push.2.push.OR
<E> ::= <E> AND <T>  ?'*'.<not>.'/'.<not>.<and>  :push.2.push.AND
<E> ::= <E> LIKE <T> ?'*'.<not>.'/'.<not>.<and>  :push.2.push.LIKE
<E> ::= NOT <E>    ?'*'.<not>.'/'.<not>.<and>    :push.1.push.NOT
<E> ::= - <E>      ?'*'.<not>.'/'.<not>.<and>    :push.1.push.~
<E> ::= <E> - <T>  ?'*'.<not>.'/'.<not>.<and>    :push.2.push.-
<E> ::= <E> + <T>  ?'*'.<not>.'/'.<not>.<and>    :push.2.push.+
<E> ::= <T>        ?'*'.<not>.'/'.<not>.<and>
<T> ::= <T> / <F>                                :push.2.push./
<T> ::= <T> * <F>                                :push.2.push.*
<T> ::= <F>        ?']'.<not>
<F> ::= <i> ( <E> )                              :push.2.push.CALL
<F> ::= [ <F> ]                                  :push.0.push.]
<F> ::= <i>          ?'='.<not>.'('.<not>.<and>  !push.I
<F> ::= ( <E> )
```