Семантический анализ

КС-грамматики, с помощью которых описывают синтаксис языков программирования, не позволяют задавать контекстные условия (КУ), имеющиеся в любом языке. Проверку КУ называют **семантическим анализом**.

Наиболее часто встречающиеся контекстные условия:

- каждый используемый в программе идентификатор должен быть описан, но не более одного раза в одной зоне описания;
- при вызове функции число и тип фактических параметров должны соответствовать числу и типам формальных параметров;
- обычно в языке накладываются ограничения на
  - типы операндов любой операции, определенной в этом языке;
  - типы левой и правой частей в операторе присваивания;
  - тип параметра цикла;
  - тип условия в операторах цикла и условном операторе и т.п.

Проверка КУ будет проводиться, как только синтаксический анализатор распознает конструкцию, на компоненты которой наложены некоторые ограничения, т.е. на этапе синтаксического анализа должны выполняться некоторые дополнительные действия, осуществляющие семантический контроль.

Вставка действий в КС-грамматику

Если для синтаксического анализа используется **метод рекурсивного спуска,** то для контроля КУ в тела процедур РС-метода необходимо вставить вызовы дополнительных "семантических" процедур (семантические действия).

Сначала семантические действия вставляются в синтаксические правила, а потом по этим расширенным правилам строятся процедуры РС-метода.

Чтобы отличать вызовы семантических процедур от других символов грамматики, они заключаются в угловые скобки.

Фактически при этом расширяется понятие КС-грамматики.

Пусть в грамматике есть правило

A → a < D1 > B < D1; D2 > | b C < D3 > , где

A, B, C ∈ N; a,b ∈ T; **<** Di **>** - есть вызов процедуры Di, i = 1, 2, 3.

По такому правилу грамматики процедуру для РС-метода будет следующей:

```
void A ( ) {
  if (c == 'a‘) { gc( ); D1( ); B( ); D1( ); D2( ); }
  else
       if (c == 'b') { gc( ); C( ); D3( ); }
        else throw c;
}
```

Пример

Написать грамматику, которая порождает язык
L = {$\alpha \in (0,1)^+\perp$ | α содержит равное количество 0 и 1}.

Решить эту задачу можно
- чисто синтаксическими средствами - описать цепочки, обладающие нужным свойством;
- с помощью синтаксических правил описать произвольные цепочки из 0 и 1, а потом вставить действия для отбора цепочек с равным количеством 0 и 1.

S → **<** k0 = k1 = 0; **>** A ⊥
A → 0 **<** k0++; **>** B | 1 **<** k1++;**>** B
B → A | ε < if (k0 != k1) throw "ERROR !!!"; >

Семантический анализатор для М-языка

Контекстные условия, выполнение которых надо контролировать в программах на М-языке, таковы:

- Любое имя, используемое в программе, должно быть описано и только один раз.
- В операторе присваивания типы переменной и выражения должны совпадать.
- В условном операторе и в операторе цикла в качестве условия возможно только логическое выражение.
- Операнды операции отношения должны быть целочисленными.
- Тип выражения и совместимость типов операндов в выражении определяются по обычным правилам (как в Паскале).

Для проверки КУ М-языка в синтаксические правила грамматики вставим вызовы процедур, осуществляющих необходимый контроль, а затем перенесем их в процедуры рекурсивного спуска.

Обработка описаний

В синтаксические правила для описаний нужно вставить действия, с помощью которых можно запомнить **типы** переменных и контролировать **единственность их описания**.

i-ая строка таблицы TID соответствует идентификатору-лексеме вида (LEX_ID, i).

Лексический анализатор заполнил поле **name**; значения полей **declare** и **type** будем заполнять на этапе семантического анализа.

Раздел описаний имеет вид

D → I { ,I }: [int | bool],

т.е. имени типа (int или bool) предшествует список идентификаторов.

Эти идентификаторы (номера соответствующих им строк таблицы TID) надо запомнить, например, в стеке целых чисел

Stack < int, 100 > st_int ,

а когда будет проанализировано имя типа, надо заполнить поля declare и type в соответствующих строках.

Функция ***void Parser::dec (type_of_lex type)*** :

- ✓ считывает из стека номера строк таблицы TID,
- ✓ заносит в них информацию о типе соответствующих переменных и о наличии их описаний и
- ✓ контролирует повторное описание переменных.

```
void Parser::dec ( type_of_lex type ) {
        int i;
        while ( ! st_int.is_empty ( )) {
                i = st_int.pop ();
                if ( TID [ i ].get_declare () ) throw "twice";
                else {
                        TID [ i ].put_declare ();
                        TID [ i ].put_type (type);
                }
        }
}
```

С учетом имеющихся функций правило вывода с действиями для обработки описаний будет таким:

D → **<** st_int.reset () **>** I **<** st_int.push (c_val) **>**
{ , I **<** st_int.push (c_val) **>** }:
[int **<** dec (LEX_INT) **>** | bool **<** dec (LEX_BOOL) **>**]

Контроль контекстных условий в выражении

Типы операндов и обозначение операций будем хранить в стеке ***Stack < type_of_lex, 100 > st_lex***.

Если в выражении встречается лексема-целое_число или логические константы ***true*** или ***false***, то соответствующий тип сразу заносится в стек.

Если операнд - лексема-переменная, то необходимо проверить, описана ли она; если описана, то ее тип надо занести в стек.

Эти действия выполняются с помощью функции check_id:

```
void parser::check_id ( ) {
        If (TID [ c_val ].get_declare ( ))
                st_lex.push (TID [ c_val ].get_type ( ) );
        else throw "not declared";
}
```

Для контроля контекстных условий каждой тройки - "операнд-операция-операнд" используется функцию check_op:

```
void Parser::check_op ( ) {
    type_of_lex t1, t2, op, t = LEX_INT, r = LEX_BOOL;
    t2 = st_lex.pop( );
    op = st_lex.pop( );
    t1 = st_lex.pop( );
    if (op == LEX_PLUS || op == LEX_MINUS ||op == LEX_TIMES ||op == LEX_SLASH)
        r = LEX_INT;
    if (op == LEX_OR || op == LEX_AND)
        t = LEX_BOOL;
    if (t1 == t2  &&  t1 == t) st_lex.push( r );
    else throw "wrong types are in operation";
    prog.put_lex (Lex (op) );
}
```

Для контроля за типом операнда одноместной операции *not* будем использовать функцию check_not:

```
void Parser::check_not ( ) {
    if (st_lex.pop ( ) != LEX_BOOL)
        throw "wrong type is in not";
    else {
        st_lex.push (LEX_BOOL);
        prog.put_lex (Lex (LEX_NOT));
    }
}
```

Сравним грамматики, описывающие выражения, состоящие из символов + , * , (,) , i:

G1: $E \rightarrow E+E \mid E*E \mid (E) \mid i$

G2: $E \rightarrow E+T \mid E*T \mid T$

$T \rightarrow i \mid (E)$

G3: $E \rightarrow T+E \mid T*E \mid T$

$T \rightarrow i \mid (E)$

G4: $E \rightarrow T \mid E+T$

$T \rightarrow F \mid T*F$

$F \rightarrow i \mid (E)$

G5: $E \rightarrow T \mid T+E$

$T \rightarrow F \mid F*T$

$F \rightarrow i \mid (E)$

Правила вывода выражений модельного языка с действиями для контроля контекстных условий

E → E1 | E1 [= |< |>] <st_char.push(TD[c_val])> E1<check_op()>

E1 → T { [+ | - | or] <st_char.push (TD[c_val])> T <check_op()>}

T → F { [* | / | and] <st_char.push (TD[c_val])>F<check_op()>}

F → I < check_id() > | N <st_char.push ("int")> |
[true | false] <st_char.push ("bool")> | not F < check_not() > | (E)

Контроль контекстных условий в операторах

S → I := E | if E then S else S | while E do S | B | read (I) | write (E)

Оператор присваивания I := E

Контекстное условие: в операторе присваивания типы переменной I и выражения E должны совпадать.

В результате контроля контекстных условий выражения E в стеке останется тип этого выражения (как тип результата последней операции).

При анализе идентификатора I проверяется, описан ли он, и его тип заносится в тот же стек (с помозью функции check_id()). При этом достаточно в нужный момент считать из стека два элемента и сравнить их:

```
void Parser::eq_type () {
        if ( st_lex.pop() != st_lex.pop() )
                throw "wrong types are in :=";
}
```

Правило вывода для оператора присваивания:

I **<** check_id ( ) **>** := E **<** eq_type ( ) **>**

Условный оператор, оператор цикла, оператор ввода

if E then S else S | while E do S | read (I)

Контекстные условия:

- в условном операторе и в операторе цикла в качестве условия возможны только логические выражения.
- операнд оператора ввода должен быть описан.

Для контроля КУ в **условном операторе и операторе цикла** функция eq_bool ():

```
void Parser::eq_bool () {
        if ( st_lex.pop() != LEX_BOOL )
                throw "expression is not boolean";
}
```

Для поверки операнда **оператора ввода** read (I) используется следующую функцию:

```
void Parser::check_id_in_read () {
        if ( !TID [c_val].get_declare( ) ) throw "not declared";
}
```

Правила вывода для условного оператора и операторов цикла и ввода:

if E **<** eq_bool () **>** then S else S | while E **<** eq_bool ( ) **>** do S | read (I < check_id_in_read ( )>)

Грамматика с действиями для раздела описаний М-языка

```
void Parser::D ( ) {
        st_int.reset ( );
        if (c_type != LEX_ID) throw curr_lex;
        else {
                st_int.push ( c_val );
                gl ( );
                while (c_type == LEX_COMMA) {
                        gl ( );
                        if (c_type != LEX_ID)  throw curr_lex;
                        else {
                                st_int.push ( c_val ); gl ( );
                        }
                }
                if (c_type != LEX_COLON)  throw curr_lex;
                else {    gl ( );
                        if (c_type == LEX_INT) { dec ( LEX_INT ); gl ( ); }
                        else
                        if (c_type == LEX_BOOL) { dec ( LEX_BOOL ); gl ( ); }
                        else throw curr_lex;
                }
        }
}
```