Синтаксический анализ

Написать синтаксический анализатор, обнаруживающий наибольшее число ошибок, для приведённой ниже грамматики (данная грамматика является упрощённым вариантом грамматики языка *C*):

< program >: < type > ’main’ ‘(‘ ‘)’ ‘{‘ < statement > ‘}’

< type >: ‘int’

| ‘bool’

| ‘void’

<statement>:

| < declaration > ‘;’

| ‘{‘ < statement > ‘}’

| < for > < statement >

| < if > < statement >

| < return >

< declaration >: < type > < identifier > < assign >

< identifier >: < character >< id_end >

< character >: ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i' | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’ | ‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ | ‘z’ | ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’ | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’ | ‘O’ | ‘P’ | ‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘U’ | ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’ | ‘_’

<id_end>:

| <character><id_end>

<assign>:

| ‘=’ <assign_end>

<assign_end>: <identifier>

| <number>

<number>: <digit><number_end>

<digit>: ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

<number_end>:

| <digit><number_end>

<for>: ‘for’ ‘(‘ <declaration> ‘;’ <bool_expression> ‘;’ ‘)’

<bool_expression>: <identifier> <relop> <identifier>

 | <number> <relop> <identifier>

<relop>: ‘<’ | ‘>’ | ‘==’ | ‘!=’

<if>: ‘if’ ‘(‘ <bool_expression> ‘)’

<return>: ‘return’ <number> ‘;’

Первые четыре строки – это описание main. А что идет дальше? Какие варианты кода может реализовать данный шаблон?

Простейший ручной способ синтаксического анализа – алгоритм рекурсивного спуска, который позволяет разбирать грамматики **LL(1)**. Данная грамматика как раз такая, что упрощает задачу.

Правила кодируются очень простым способом. Например, что такое *программа*?

program :: = type "main" "(" ")" "{" statement "}"

Вот как это кодируется в *C*:

```
char buffer[1024];

int program(FILE* stream)
{
 int statement_result;
 int lexem = get_lexem(stream, buffer);
 if(lexem != INT || lexem != BOOL && lexem != VOID)
  return UNRECOGNIZED_MAIN_TYPE;

 lexem = get_lexem(stream, buffer);
 if(lexem != MAIN)
  return EXPECTED_MAIN;

 lexem = get_lexem(stream, buffer);
 if(lexem != LPAREN)
  return EXPECTED_LPAREN;

 lexem = get_lexem(stream, buffer);
 if(lexem != RPAREN)
  return EXPECTED_RPAREN;

 lexem = get_lexem(stream, buffer);
 if(lexem != LBRACE)
  return EXPECTED_LBRACE;

 statement_result = statement(stream);
```

```
  if(statement_result != OK)
    return statement_result;

  lexem = get_lexem(stream, buffer);
  if(lexem != RBRACE)
    return EXPECTED_RBRACE;

  return OK;
}
```

Константы OK, UNRECOGNIZED_MAIN_TYPE, EXPECTED_MAIN необходимо определить самостоятельно. Они описывают ошибки, которые грамматика может распознать. UNRECOGNIZED_MAIN_TYPE означает, что анализатор ожидает один из трёх типов, но получил что-то другое. Константа OK означает, что ошибок нет.

Конструкция

```
lexem = get_lexem(stream, buffer);
if(lexem != RBRACE)
  return EXPECTED_RBRACE;
```

встречается очень часто. Её можно назвать "*требовать наличия лексемы во входном потоке*". Для неё удобно завести отдельные функции:

```
bool require_lexem1(FILE* stream, int expected1)
{
  int lexem = get_lexem(stream);
  return lexem == expected1;
}

bool require_lexem2(FILE* stream, int expected1, int expected2)
{
  int lexem = get_lexem(stream);
  return lexem == expected1 || lexem == expected2;
}

bool require_lexem3(FILE* stream, int expected1, int expected2, int expected3)
{
  int lexem = get_lexem(stream);
  return lexem == expected1 || lexem == expected2 || lexem == expected3;
}
```

Теперь функцию program можно сделать проще и короче:

```
int program(FILE* stream)
{
```

```
  int statement_result;

  if(!require_lexem3(stream, INT, BOOL, VOID))
    return UNRECOGNIZED_MAIN_TYPE;

  if(!require_lexem1(stream, MAIN))
    return EXPECTED_MAIN;

  if(!require_lexem1(stream, LPAREN))
    return EXPECTED_LPAREN;

  if(!require_lexem1(stream, RPAREN))
    return EXPECTED_RPAREN;

  if(!require_lexem1(stream, LBRACE))
    return EXPECTED_LBRACE;

  statement_result = statement(stream);
  if(statement_result != OK)
    return statement_result;

  if(!require_lexem1(stream, RBRACE))
    return EXPECTED_RBRACE;

  return OK;
}
```

Функция statement сложнее. Она на основании следующей лексемы принимает решение о том, какое из альтернативных правил применить.

```
int statement(FILE* stream)
{
  int statement_result;
  int lexem = get_lexem(stream);

  if(lexem == INT || lexem == BOOL || lexem == VOID)
    return declaration(stream);

  if(lexem == LBRACE) {
    statement_result = statement(stream);

    if(statement_result != OK)
      return statement_result;
```

```
  if(!require_lexem1(stream, RBRACE))
    return EXPECTED_RBRACE;

  return OK;
 }

 if(lexem == FOR)
  return for_rule(stream);

 if(lexem == IF)
  return if_rule(stream);

 if(lexem == RETURN)
  return return_rule(stream);

 return UNRECOGNIZED_STATEMENT;
}
```

Эта функция рекурсивна, поэтому позволяет распознавать сложные конструкции вида { { { int i = 300; } } }. Для примера реализуем также функцию declaration:

```
int declaration(FILE* stream)
{
 if(!require_lexem1(stream, IDENTIFIER))
  return EXPECTED_IDENTIFIER;

 if(!require_lexem1(stream, ASSIGN))
  return EXPECTED_ASSIGN;

 if(!require_lexem2(stream, IDENTIFIER, NUMBER))
  return EXPECTED_IDENTIFIER_OR_NUMBER;

 if(!require_lexem1(stream, SEMICOLON))
  return EXPECTED_SEMICOLON;

 return OK;
}
```

Стараемся придерживаться принципа: для каждого правила в грамматике писать отдельную функцию, так удобнее проводить соответствия. В приведённом коде отошли от этого принципа, чтобы сделать код чуть короче, в частности, правила assign и assign_end попали внутрь declaration.

В результате алгоритм спускается от верхнего общего правила к нижним (от program к declaration через statement), при этом некоторые функции могут вызывать друг друга рекурсивно.

Именно поэтому алгоритм называется рекурсивным спуском.

Реализовав все правила, получим полноценный анализатор программ, который сможет сообщать нам об ошибках.

Напишем предпоследнюю функцию, которая превратит коды ошибок в текстовые сообщения:

```
const char* get_message(int code)
{
  switch(code) {
  case OK:
    return "Ok";

  case UNRECOGNIZED_MAIN_TYPE:
    return "Unrecognized main type";

  . . .
  }

  return "Unrecognized code";
}
```

Главная функция программы оказывается очень простой:

```
void main()
{
  int code = program(stdin);
  const char* message = get_message(code);
  printf("%s\n", message);
}
```

Заключение.

Был реализован лексический анализатор (get_lexem), набор вспомогательных функций для проверки наличия лексем (require_lexem1, require_lexem2, require_lexem3) и набор функций для каждого правила грамматики (program, statement, declaration и множество других, которые вам предстоит написать самостоятельно). Функция get_message позволила выводить понятный текст вместо непонятных кодов.

Получилось два набора констант: коды лексем (LPAREN, IDENTIFIER и т.д.) и коды ошибок (OK, EXPECTED_ASSIGN и т.д.)

Вся программа целиком способна проверить код другой программы и вывести OK, если она соответствует грамматике. В противном случае она выводит сообщение об ошибке.