Курс «Компиляторные технологии»

Лекция 4

Синтаксически управляемая трансляция (3)

Курносов Михаил Георгиевич

www.mkurnosov.net

Сибирский государственный университет телекоммуникаций и информатики
Весенний семестр

Лексический анализатор для транслятора простых выражений

- **Лексический анализатор** считывает символы из входной строки и группирует их в токены (token) — терминал с дополнительными атрибутами (тип токена, лексема)
- **Расширение грамматики:** идентификаторы, целочисленные литералы

$$
\begin{array}{lcll}
expr & \rightarrow & expr + term & \{ \text{print}('+') \} \\
 & | & expr - term & \{ \text{print}('-') \} \\
 & | & term & \\
 & & & \\
term & \rightarrow & term * factor & \{ \text{print}('*') \} \\
 & | & term / factor & \{ \text{print}('/') \} \\
 & | & factor & \\
 & & & \\
factor & \rightarrow & (expr) & \\
 & | & \textbf{num} & \{ \text{print}(\textbf{num}.value) \} \\
 & | & \textbf{id} & \{ \text{print}(\textbf{id}.lexeme) \}
\end{array}
$$

- **num**.*value* — атрибут терминала **num** со значением целочисленного литерала в десятичной системе исчисления
- **id**.*lexeme* — строковый атрибут терминала **id** со значением идентификатора

Лексический анализатор: пропуск пробельных символов

```
Token Lexer::nextToken()
{
    // Пропуск пробельных символов
    for ( ; ; peek = readNextChar()) {
      if (peek == ' ' || peak == '\t')
          continue;
      else if (peek == '\n')
          line = line + 1;
      else
          break;
    }

    // ...

    return Token(tokenType, lexeme)
}
```

Лексический анализатор: опережающее чтение

- **Распознавание лексем с общим префиксом**: `>`, `>=`, `>>`, ...
- Лексическому анализатору может требоваться прочесть несколько символов за текущим, чтобы решить, какой именно токен следует вернуть
- **Опережающее чтение** (reading ahead) — поддержка входного буфера, из которого анализатор может выполнять чтение и в который может возвращать прочитанные символы
- Альтернатива — буфер с *N* вперед прочитанными символами + указатель на текущий символ

Целочисленные литералы (константы)

- **Вход анализатора**: 31 + 28 + −59
- **Выход**: <num, 31> <plus> <num, 28> <plus> <minus> <num, 59>

```
{

    // Целочисленный литерал в десятичной системе исчисления
    if (isDigit(peek))
        value = 0
        do {
            value = value * 10 + strToInt(peek)   // Целочисленное значение цифры
            peak = readNextChar();
        } while (isDigit(peek))
        return Token(NUM, value)
    }

}

bool isDigit(char ch)
{
    return ch == '0' || ch == '1' || ... || ch == '9';
}
```

Clang

- https://clang.llvm.org/doxygen/Lexer_8cpp_source.html

```
2028 /// LexNumericConstant - Lex the remainder of a integer or floating point
2029 /// constant. From[-1] is the first character lexed.  Return the end of the
2030 /// constant.
2031 bool Lexer::LexNumericConstant(Token &Result, const char *CurPtr) {
2032   unsigned Size;
2033   char C = getCharAndSize(CurPtr, Size);
2034   char PrevCh = 0;
2035   while (isPreprocessingNumberBody(C)) {
2036     CurPtr = ConsumeChar(CurPtr, Size, Result);
2037     PrevCh = C;
2038     if (LangOpts.HLSL && C == '.' && (*CurPtr == 'x' || *CurPtr == 'r')) {
2039       CurPtr -= Size;
2040       break;
2041     }
2042     C = getCharAndSize(CurPtr, Size);
2043   }
2044
2045   // If we fell out, check for a sign, due to 1e+12.  If we have one, continue.
2046   if ((C == '-' || C == '+') && (PrevCh == 'E' || PrevCh == 'e')) {
2047     // If we are in Microsoft mode, don't continue if the constant is hex.
2048     // For example, MSVC will accept the following as 3 tokens: 0x1234567e+1
2049     if (!LangOpts.MicrosoftExt || !isHexaLiteral(BufferPtr, LangOpts))
2050       return LexNumericConstant(Result, ConsumeChar(CurPtr, Size, Result));
2051   }
2052
2053   // If we have a hex FP constant, continue.
2054   if ((C == '-' || C == '+') && (PrevCh == 'P' || PrevCh == 'p')) {
2055     // Outside C99 and C++17, we accept hexadecimal floating point numbers as a
2056     // not-quite-conforming extension. Only do so if this looks like it's
2057     // actually meant to be a hexfloat, and not if it has a ud-suffix.
2058     bool IsHexFloat = true;
2059     if (!LangOpts.C99) {
2060       if (!isHexaLiteral(BufferPtr, LangOpts))
2061         IsHexFloat = false;
2062       else if (!LangOpts.CPlusPlus17 &&
2063                std::find(BufferPtr, CurPtr, '_') != CurPtr)
2064         IsHexFloat = false;
2065     }
2066     if (IsHexFloat)
2067       return LexNumericConstant(Result, ConsumeChar(CurPtr, Size, Result));
2068   }
```

Распознавание ключевых слов и идентификаторов

- **Ключевое слово** (keyword) — фиксированные символьные строки для обозначения языковых конструкций или синтаксических знаков препинания
- **Примеры**: for, do, while, if, else, then
- Грамматики обычно трактуют идентификаторы как терминалы для упрощения синтаксического анализатора
- Два варианта
  1. Ключевые слова зарезервированы, идентификаторы не могут совпадать с ключевыми словами
  2. Идентификаторы могут совпадать с ключевыми словами
     (APL, PL/I, Fortran: https://fortranwiki.org/fortran/show/Keywords)

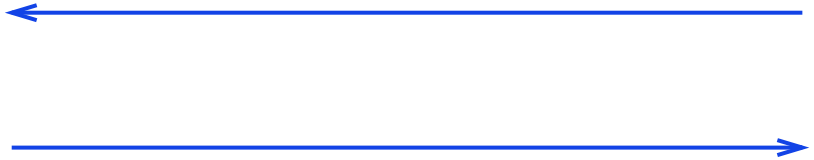
Хранение ключевых слов и идентификаторов в одной таблице символов

```
{
    if (isAlpha(peek)) {
        // Собираем буквы или цифры в буфер str
        // Предполагается, что строка str должна быть максимально
        // возможной длины, анализатор продолжает чтение до тех пор,
        // пока считываются буквы и цифры
        str = peek
        for (peek = readNextchar(); isAlphaNum(peek); peek = readNextchar())
             str += peek

        token = symbolTable.lookup(str)
        if (!token) {
            token = Token(IDENT, str)
            symbolTable.add(str, token)
        }
        return token;
    }

}

bool isAlpha(char ch)
{
    return ch == 'a' || ch == 'A' || ch == 'b' || ... || ch == 'Z';
}
```



Ключевые слова заранее внесены в таблицу символов (symbol table)

#	Symbol	Token
1	**if**	(KW_IF, 0)
2	**then**	(KW_THEN, 0)
3	**for**	(KW_FOR, 0)
.	...	...
78	nUsers	(IDENT, 'nUsers')
79	sum	(IDENT, 'sum')

Структура основной функции лексического анализатора

```
Token Lexer::nextToken()
{
    skipWhitespaces()           // Пропуск пробельных символов
    parseNumber()               // Обработка чисел
    parseKeywordsAndIdents()    // Обработка зарезервированных слов и идентификаторов

    Token token = Token(peek)   // Считанный символ peek выступает как токен: (, ), +, %, ...
    peek = ' '                  // Инициализация пробелом для продолжения обработки потока
    return token
}
```

Таблица символов

- **Таблица символов** (symbol table) — структура данных, которая хранит информацию о конструкциях исходной программы
- Таблица заполняется и модифицируется инкрементно на начальной стадии работы компилятора
- Записи в таблице символов:
  - информацию об идентификаторах
  - местоположение в памяти
- Поддержка множественных объявлений одного и того же идентификатора (отдельная таблица символов для каждой области видимости)
- Операции:
  - Добавление символа с указанием типа токена
  - Поиск по символу

Область видимости (scope)

```
{
  int x; char y;     // Объявление x, y

  {
    bool y;
    x; y;            // Объявление y, использование x:int, y:bool
  }

  x; y;              // Использование x:int, y:char
}
```

- **Правило последнего вложения** (most-closely nested) для блоков — идентификатор *x* находится в области видимости последнего по вложенности объявления *x*
- Определение идентификатора следует искать путем перебора блоков изнутри наружу (снизу вверх), начиная с блока, в котором находится интересующий идентификатор
- **Реализация таблиц символов для вложенных блоков:**
  - **Стек локальных таблиц символов**: на вершине стека находится таблица для текущего блока, ниже в стеке располагаются таблицы охватывающих блоков
  - **Единая хеш-таблица**: при выходе из блока компилятор должен отменить все изменения, внесенные в хеш-таблицу объявлениями в блоке (вспомогательный стек для отслеживания изменений в хеш-таблице при обработке блока)

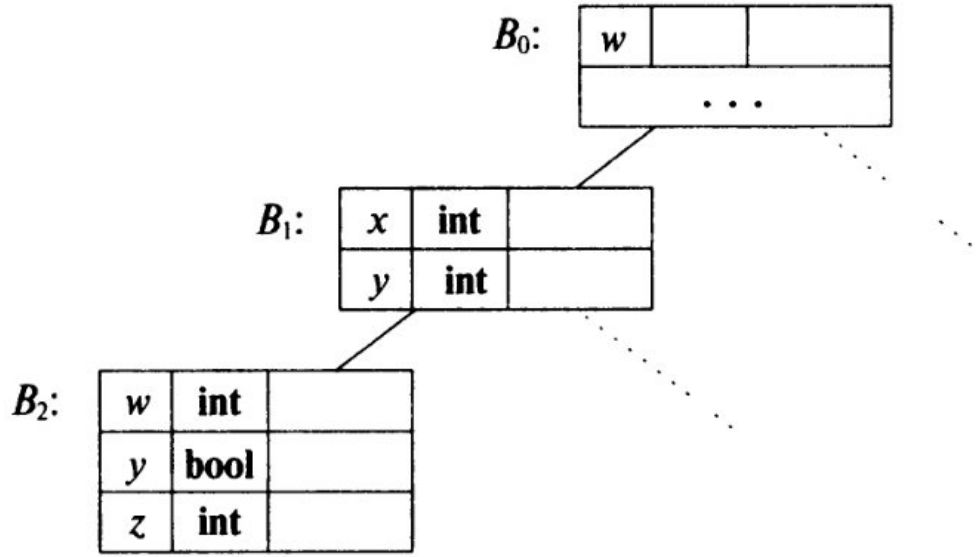
Области видимости

```
1)  {    int x1; int y1;
2)       {    int w2; bool y2; int z2;
3)            ··· w2 ···; ··· x1 ···; ··· y2 ···; ··· z2 ···;
4)       }
5)        ··· w0 ···; ··· x1 ···; ··· y1 ···;
6)  }
```

Два вложенных блока:

- Блок 1: x_1, y_1,
- Блок 2: w_2, y_2, z_2
- Блок 0 (внешний): w_0

- Реализация правила ближайшего вложенного блока на базе цепочек таблиц символов – таблица для вложенного блока указывает на таблицу для охватывающего блока (outer)



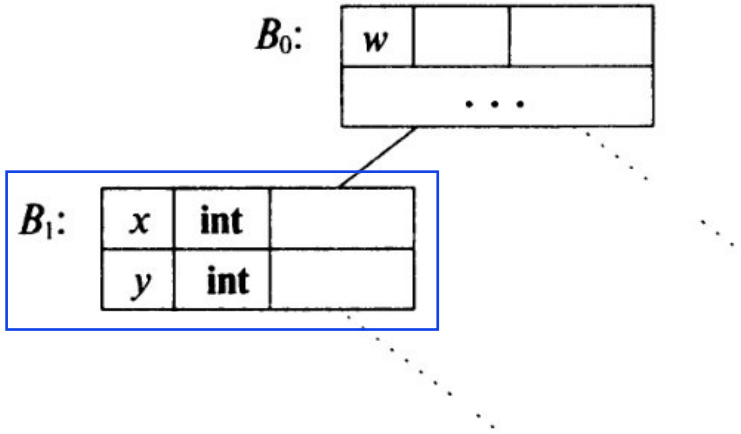
Области видимости

```
1)  {    int x1; int y1;                                  Создание B1
2)       {    int w2; bool y2; int z2;
3)            ··· w2 ···; ··· x1 ···; ··· y2 ···; ··· z2 ···;
4)       }
5)        ··· w0 ···; ··· x1 ···; ··· y1 ···;
6)  }
```

Два вложенных блока:

- Блок 1: x_1, y_1,
- Блок 2: w_2, y_2, z_2
- Блок 0 (внешний): w_0

- Реализация правила ближайшего вложенного блока на базе цепочек таблиц символов – таблица для вложенного блока указывает на таблицу для охватывающего блока (outer)



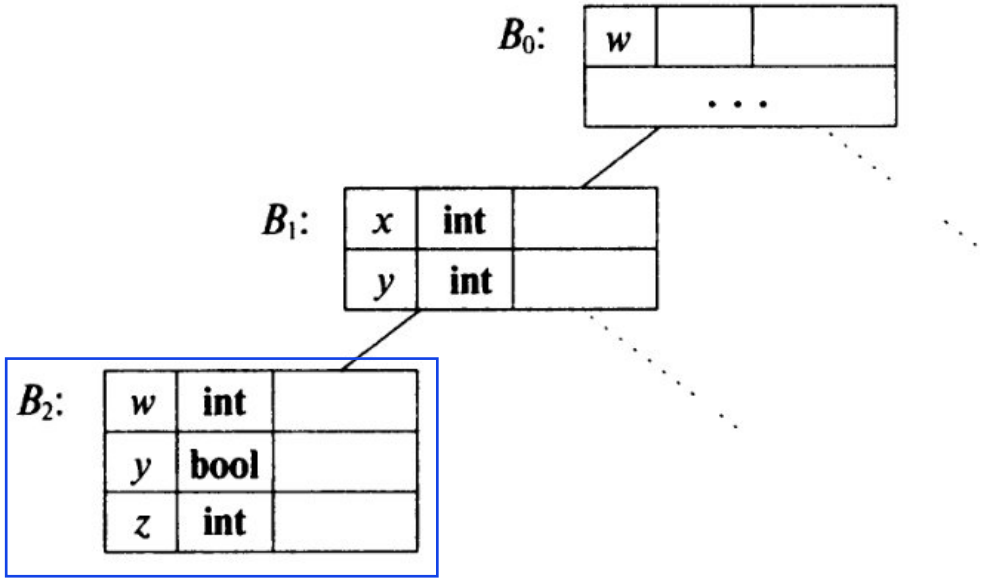
Области видимости

```
1)  {   int x1; int y1;
2)      {   int w2; bool y2; int z2;          Создание B2
3)          ··· w2 ···; ··· x1 ···; ··· y2 ···; ··· z2 ···;
4)      }
5)      ··· w0 ···; ··· x1 ···; ··· y1 ···;
6)  }
```

Два вложенных блока:

- Блок 1: x_1, y_1,
- Блок 2: w_2, y_2, z_2
- Блок 0 (внешний): w_0

- Реализация правила ближайшего вложенного блока на базе цепочек таблиц символов – таблица для вложенного блока указывает на таблицу для охватывающего блока (outer)



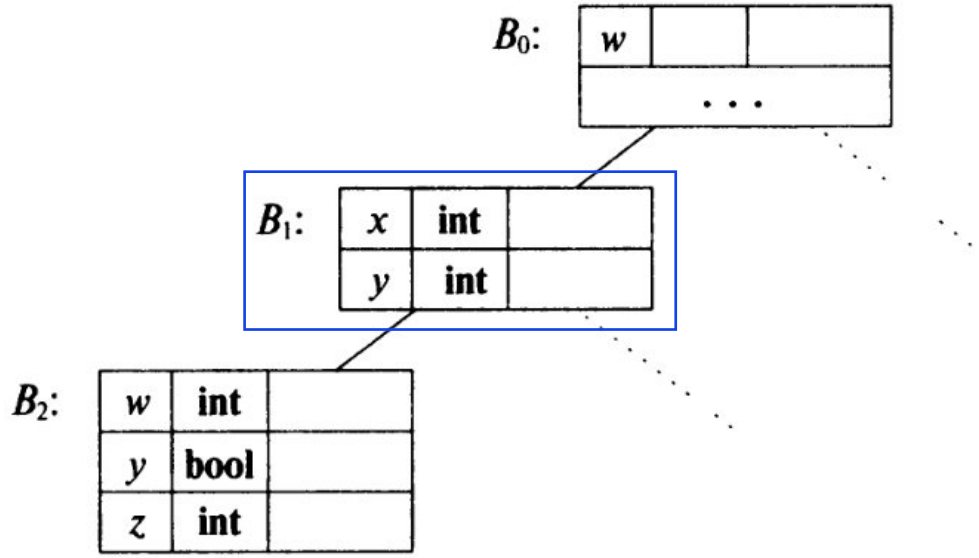
Области видимости

```
1)  {   int x1; int y1;
2)      {   int w2; bool y2; int z2;
3)          ··· w2 ···; ··· x1 ···; ··· y2 ···; ··· z2 ···;
4)      }
5)      ··· w0 ···; ··· x1 ···; ··· y1 ···;   Переключение на B1
6)  }
```

Два вложенных блока:

- Блок 1: x_1, y_1,
- Блок 2: w_2, y_2, z_2
- Блок 0 (внешний): w_0

- Реализация правила ближайшего вложенного блока на базе цепочек таблиц символов – таблица для вложенного блока указывает на таблицу для охватывающего блока (outer)



Реализация цепочки таблиц символов

```
package symbols;
import java.util.*;
public class Env {
   private Hashtable table;
   protected Env prev;

   public Env(Env p) {
      table = new Hashtable(); prev = p;
   }

   public void put(String s, Symbol sym) {
      table.put(s, sym);
   }

   public Symbol get(String s) {
      for( Env e = this; e != null; e = e.prev ) {
         Symbol found = (Symbol)(e.table.get(s));
         if( found != null ) return found;
      }
      return null;
   }
}
```

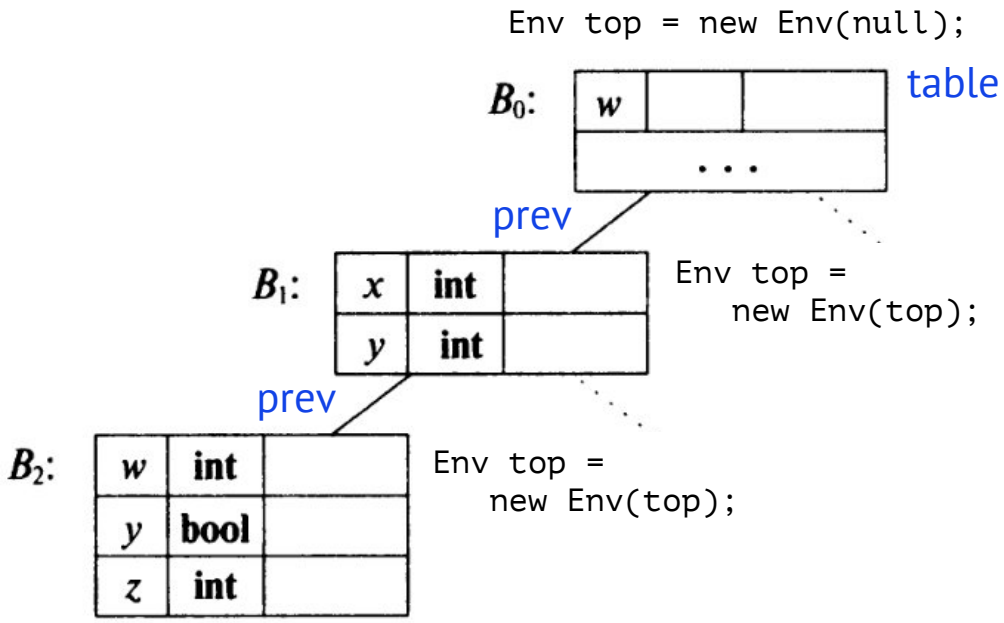


Схема трансляции с использованием таблиц символов

program	→	*block*	{ *top* = **null**; }
block	→	'{'	{ *saved* = *top*; *top* = **new** *Env*(*top*); print("{ "); }
		decls stmts '}'	{ *top* = *saved*; print("} "); }
decls	→	*decls decl*	
	|	ϵ	
decl	→	**type id** ;	{ *s* = **new** *Symbol*; *s.type* = **type**.*lexeme* *top.put*(**id**.*lexeme*, *s*); }
stmts	→	*stmts stmt*	
	|	ϵ	
stmt	→	*block*	
	|	*factor* ;	{ print("; "); }

factor	→	**id**	{ *s* = *top.get*(**id**.*lexeme*);
			print(**id**.*lexeme*);
			print(" : ");
			print(*s.type*); }

Создание новой таблицы

Переключение на предыдущую таблицу

Добавление имени переменной в текущую таблицу

Поиск имени в цепочке таблиц символов

```
{
  int x; char y;
  {
    bool y; x; y;
  }
  x; y;
}
```

Схема трансляции удаляет объявления и выдает строку — связывает использование переменных с определениями (отыскивает в таблицах символов)

```
{
  {
    x:int; y:bool;
  }
  x:int; y:char;
}
```

Генерация промежуточного кода

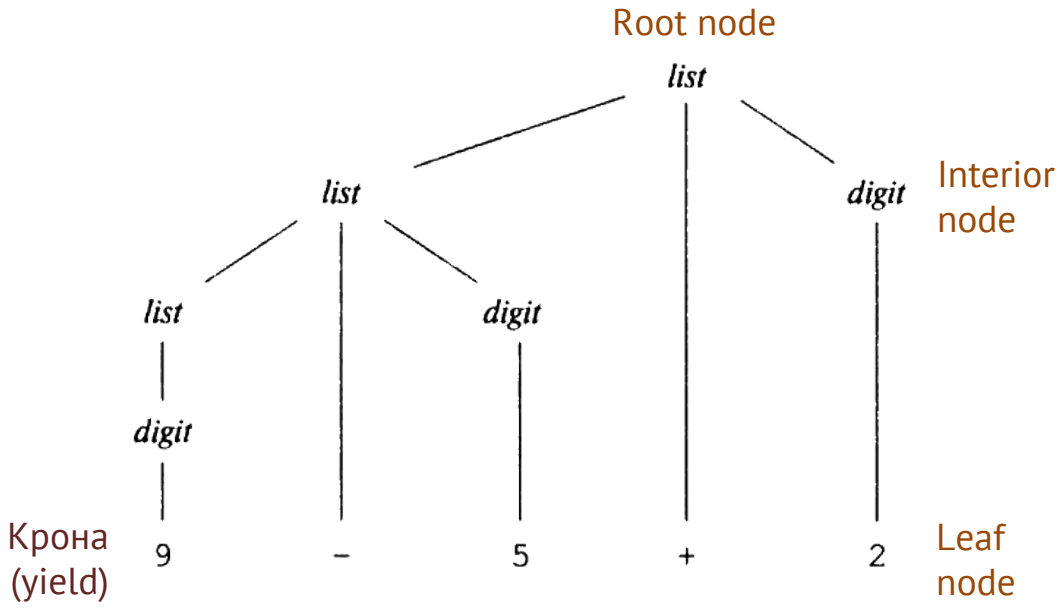
- **Два вида промежуточных представлений** (intermediate representation — IR)
  - деревья разбора, абстрактные синтаксические деревья
  - линейные представления — трехадресный код
- Компилятор может строить синтаксическое дерево одновременно с генерацией трехадресного кода

Дерево разбора —> Синтаксическое дерево —> Трехадресный код

Абстрактное синтаксическое дерево

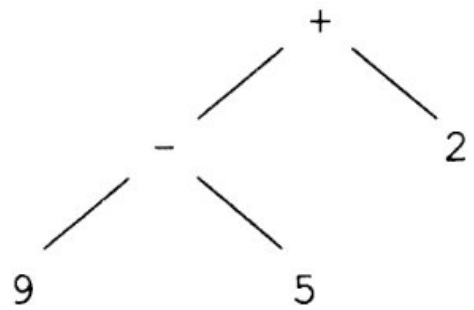
Дерево разбора (parse tree) — древовидное представление порождения строки языка из стартового символа грамматики

Дерево разбора для строки «9-5+2»



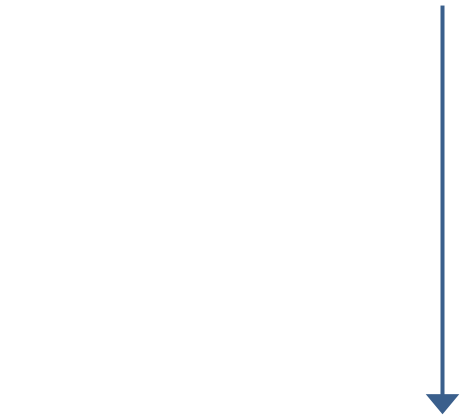
Абстрактное синтаксическое дерево (abstract syntax tree) — древовидное представление порождения строки языка, в котором узлами являются программные конструкции

Абстрактное синтаксическое дерево разбора для строки «9-5+2»



Построение синтаксических деревьев для выражений и инструкций

Приоритет (ниже)



Приоритет (выше)

=	**assign**	(правоассоциативный)
||	**cond**	
&&	**cond**	
== !=	**rel**	
< <= >= >	**rel**	
+ -	**op**	
* / %	**op**	
!	**not**	
$-_{unary}$	**minus**	
[ ]	**access**	

Построение синтаксических деревьев для выражений и инструкций

```
program → block                 { return block.n; }

block → '{' stmts '}'           { block.n = stmts.n; }

stmts → stmts1 stmt             { stmts.n = new Seq(stmts1.n, stmt.n); }
      | ϵ                       { stmts.n = null; }

stmt → expr ;                   { stmt.n = new Eval(expr.n); }
     | if ( expr ) stmt1
                                { stmt.n = new If(expr.n, stmt1.n); }
     | while ( expr ) stmt1
                                { stmt.n = new While(expr.n, stmt1.n); }
     | do stmt1 while ( expr ) ;
                                { stmt.n = new Do(stmt1.n, expr.n); }
     | block                    { stmt.n = block.n; }

expr → rel = expr1              { expr.n = new Assign('=', rel.n, expr1.n); }
     | rel                      { expr.n = rel.n; }

rel → rel1 < add                { rel.n = new Rel('<', rel1.n, add.n); }
    | rel1 <= add               { rel.n = new Rel('≤', rel1.n, add.n); }
    | add                       { rel.n = add.n; }

add → add1 + term               { add.n = new Op('+', add1.n, term.n); }
    | term                      { add.n = term.n; }

term → term1 * factor           { term.n = new Op('*', term1.n, factor.n); }
     | factor                   { term.n = factor.n; }

factor → ( expr )               { factor.n = expr.n; }
       | num                    { factor.n = new Num(num.value); }
```

```
stmt → block                    { stmt.n = block.n; }
block → '{' stmts '}'           { block.n = stmts.n; }
```

- Первое правило — если инструкция представляет собой блок, то она имеет то же синтаксическое дерево, что и блок
- Второе правило — синтаксическое дерево для нетерминала block представляет собой просто синтаксическое дерево для последовательности инструкций в блоке

Построение синтаксических деревьев для выражений и инструкций

$$
\begin{array}{rcll}
\textit{program} & \rightarrow & \textit{block} & \{ \text{return } \textit{block.n}; \} \\
\textit{block} & \rightarrow & \text{'\{'} \ \textit{stmts} \ \text{'\}'} & \{ \textit{block.n} = \textit{stmts.n}; \} \\
\textit{stmts} & \rightarrow & \textit{stmts}_1 \ \textit{stmt} & \{ \textit{stmts.n} = \textbf{new } \textit{Seq}\,(\textit{stmts}_1.n, \textit{stmt.n}); \} \\
 & | & \epsilon & \{ \textit{stmts.n} = \textbf{null}; \} \\
\textit{stmt} & \rightarrow & \textit{expr} \ ; & \{ \textit{stmt.n} = \textbf{new } \textit{Eval}\,(\textit{expr.n}); \} \\
 & | & \textbf{if} \ (\ \textit{expr}\ ) \ \textit{stmt}_1 & \{ \textit{stmt.n} = \textbf{new } \textit{If}\,(\textit{expr.n}, \textit{stmt}_1.n); \} \\
 & | & \textbf{while} \ (\ \textit{expr}\ ) \ \textit{stmt}_1 & \{ \textit{stmt.n} = \textbf{new } \textit{While}\,(\textit{expr.n}, \textit{stmt}_1.n); \} \\
 & | & \textbf{do } \textit{stmt}_1 \ \textbf{while} \ (\ \textit{expr}\ ) \ ; & \{ \textit{stmt.n} = \textbf{new } \textit{Do}\,(\textit{stmt}_1.n, \textit{expr.n}); \} \\
 & | & \textit{block} & \{ \textit{stmt.n} = \textit{block.n}; \} \\
\textit{expr} & \rightarrow & \textit{rel} = \textit{expr}_1 & \{ \textit{expr.n} = \textbf{new } \textit{Assign}\,(\text{'='}, \textit{rel.n}, \textit{expr}_1.n); \} \\
 & | & \textit{rel} & \{ \textit{expr.n} = \textit{rel.n}; \} \\
\textit{rel} & \rightarrow & \textit{rel}_1 < \textit{add} & \{ \textit{rel.n} = \textbf{new } \textit{Rel}\,(\text{'<'}, \textit{rel}_1.n, \textit{add.n}); \} \\
 & | & \textit{rel}_1 <= \textit{add} & \{ \textit{rel.n} = \textbf{new } \textit{Rel}\,(\text{'}\leq\text{'}, \textit{rel}_1.n, \textit{add.n}); \} \\
 & | & \textit{add} & \{ \textit{rel.n} = \textit{add.n}; \} \\
\textit{add} & \rightarrow & \textit{add}_1 + \textit{term} & \{ \textit{add.n} = \textbf{new } \textit{Op}\,(\text{'+'}, \textit{add}_1.n, \textit{term.n}); \} \\
 & | & \textit{term} & \{ \textit{add.n} = \textit{term.n}; \} \\
\textit{term} & \rightarrow & \textit{term}_1 * \textit{factor} & \{ \textit{term.n} = \textbf{new } \textit{Op}\,(\text{'*'}, \textit{term}_1.n, \textit{factor.n}); \} \\
 & | & \textit{factor} & \{ \textit{term.n} = \textit{factor.n}; \} \\
\textit{factor} & \rightarrow & (\ \textit{expr}\ ) & \{ \textit{factor.n} = \textit{expr.n}; \} \\
 & | & \textbf{num} & \{ \textit{factor.n} = \textbf{new } \textit{Num}\,(\textbf{num}.\textit{value}); \}
\end{array}
$$

Часть синтаксического дерева для списка инструкций, состоящего из инструкций **if** и **while**

- Все нетерминалы в схеме трансляции имеют атрибут n — узел синтаксического дерева (объект класса Node)
- Класс Seq — последовательность инструкций
- Класс Node имеет два подкласса:
  - Expr — для выражений всех видов
  - Stmt — для всех видов инструкций (подклассы If, While, Do)

Статические проверки (static checking)

- На этапе лексического и синтаксического анализа выполняются статические проверки (static checking)
- **Проверка синтаксиса** (syntactic checking) — уникальность идентификаторов в области видимости, расположение инструкций в допустимых местах (break только в цикле или switch) — эти требования не покрываются грамматикой
- **Проверка типов** (type checking) — проверка корректности применения оператора/функции к корректному числу операндов допустимых и совместимых типов; преобразования типов
  (добавление соответствующего оператора в синтаксическое дерево: intToFloat, shortToInt)

```
if (a > 4) i = 3    // «a > 4» – должно быть типа boolean
if (3)  i = 3       // «3», intToBool(int) – должно быть типа boolean
```

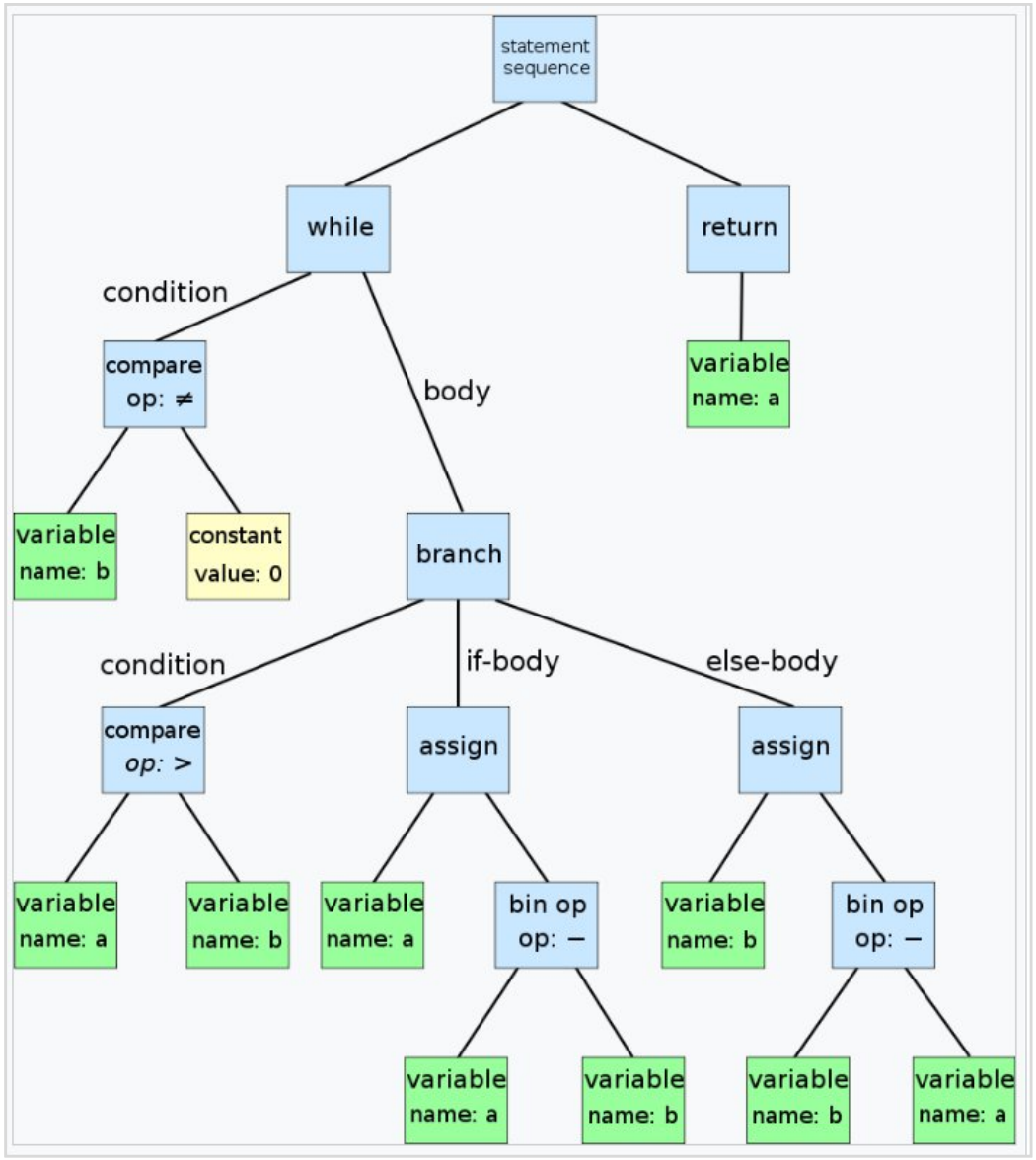
  - **Приведения** (coercion): `3 * 2.17  ==> intToDouble(3) * 2.17`
    (Определение языка указывает, какие именно приведения допустимы)
  - **Перегрузка** (overloading): поиск операции для операндов

- **Проверка L-значений и R-значений**: левая сторона оператора присваивания имеет L-значение (l-value)

```
i = 5              // l-value = r-value
i = i + 1          //
4 = i              // r-value = l-value
```

Абстрактное синтаксическое дерево

```
while b ≠ 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```



Варианты дальнейшего анализа и синтеза после построения синтаксического дерева (AST):

- генерация кода для целевой системы
- генерация кода в линейное промежуточное представления (линеаризация)

Трёхадресный код
(three-address instructions)

Арифметико-логические команды и команды доступа к памяти

- `x = y op z` `// x, y, z – константы или имена переменных`
- `x[y] = z` `// запись z в элемент y массива x`
- `x = y[z]` `// чтение в x элемент z массива y`
- `x = y` `// копирование y в x`

Команды изменения потока управления (control flow)

- **`ifFalse`** `x` **`goto`** `L` `// Если x ложно, следующей выполняется команда с меткой L`
- **`ifTrue`** `x` **`goto`** `L` `// Если x истинно, следующей выполняется команда с меткой L`
- **`goto`** `L` `// Следующей выполняется команда с меткой L`
- `L: x` `// Указание метки команде x`

Трансляция инструкций: if

```
if (expr) stmt1
```

```
Код вычисления expr во временную переменную cond
ifFalse cond goto AFTER
Код stmt1
AFTER:
```

```
class If extends Stmt {
    Expr E; Stmt S;
    public If(Expr x, Stmt y) { E = x; S = y; after = newlabel(); }
    public void gen() {
        Expr n = E.rvalue();
        emit( "ifFalse " + n.toString() + " goto " + after);
        S.gen();
        emit(after + ":");
    }
}
```

- Конструктор If создает узел синтаксического дерева
- Функция gen вызывается для генерации трехадресного кода
- После построение всего синтаксического дерева в его корне вызывается функция gen

Трансляция инструкций: выражения

```
x = y * 2
x = y op z
x[y] = z
x = y[z]
x = y
```

- Для констант и идентификаторов не генерируется никакой код — они входят в команды в качестве адресов

- Если узел x класса Expr содержит оператор op, то генерируется команда для вычисления значения в узле x во "временное" имя t

```
i – j + k
```

→

```
t1 = i - j
t2 = t1 - k
```

- При обращении к массивам и присваиваниях требуется различать l-значения и r-значения
- Выражение 2 * a[i] может быть транслировано путем вычисления r-значения a[i] во временную переменную

```
2 * a[i]
```

→

```
t1 = a[i]
t2 = 2 * t1
```

- Нельзя использовать временную переменную вместо a[i], если a[i] появляется в левой части присваивания

Трансляция инструкций: выражения

- Функция lvalue() — генерирует команды для вычисления поддеревьев ниже *x* и возвращает узел, представляющий "адрес" x

```
Expr lvalue(x : Expr) {
    if ( x является узлом Id ) return x;
    else if ( x является узлом Access (y, z), а y — узел Id ) {
        return new Access (y, rvalue(z));
    }
    else error;
}
```

```
a[2 * k]
```

```
t = 2 * k
a[t]
```

Трансляция инструкций: выражения

- Функция rvalue() — генерирует команды для вычисления *x* во временную переменную и возвращает новый узел, представляющий эту переменную

```
Expr rvalue(x : Expr) {
    if ( x — узел Id или Constant ) return x;
    else if ( x — узел Op (op, y, z) или Rel (op, y, z) ) {
        t = новая временная переменная;
        Генерация строки для t = rvalue(y) op rvalue(z);
        return Новый узел t;
    }
    else if ( x узел Access (y, z) ) {
        t = новая временная переменная;
        Вызов lvalue(x) возвращающий Access (y, z');
        Генерация строки для t = Access (y, z');
        return Новый узел t;
    }
    else if ( x — узел Assign (y, z) ) {
        z' = rvalue(z);
        Генерация строки для lvalue(y) = z';
        return z';
    }
}
```

```
a[i] = 2 * a[j - k]
```

```
t3 = j - k
t2 = a[t3]
t1 = 2 * t2
a[i] = t1
```