Курс «Компиляторные технологии»

Лекция 8

Разбор практики: AST

Курносов Михаил Георгиевич

www.mkurnosov.net

Сибирский государственный университет телекоммуникаций и информатики
Весенний семестр

Лексический анализатор

Заполнение таблиц символов в правилах Flex

Структуры данных для хранения таблицы символов

Интерфейс с синтаксическим анализатором (Bison)

Flex с таблицей символов

CoolLexer.flex

```
%{
#include "CoolLexer.h"
#include "CoolParser.h"
#include "CoolSymbolTable.h"

#undef YY_DECL
#define YY_DECL int CoolLexer::yylex()

extern YYSTYPE cool_yylval;

%}

%option warn nodefault batch noyywrap c++
%option yylineno
%option yyclass="CoolLexer"

ws              [ \t\f\r\v]*
ident           [_a-zA-Z][_a-zA-Z0-9]*
num             [0-9]+

%Start STR
%Start COMMENT_INLINE
%Start COMMENT

%%
```

CoolParser.h

```
using Boolean = bool;
class Entry;
using Symbol = Entry *;

typedef union {
    Boolean boolean;
    Symbol symbol;
    char *error_msg;
} YYSTYPE;
```

driver.cpp

```
...
YYSTYPE cool_yylval;

int main(int argc, char** argv)
{
    std::ifstream ifs(argv[1]);
    if (ifs.fail()) {
        std::cerr << "Error opening file `" << argv[1] << "`\n";
        std::exit(EXIT_FAILURE);
    }

    CoolLexer* lexer = new CoolLexer(ifs, std::cout);
    for (int token = lexer->yylex(); token; token = lexer->yylex())
{
        std::cout << "Token: " << token << " '"
                  << lexer->YYText() << "'\n";
    }

    // Dump symbol tables
    std::cerr << "Integer lexeme symbol table:\n";
    IntTable.print();

    std::cerr << "String lexeme symbol table:\n";
    StrTable.print();

    std::cerr << "Identifier symbol table:\n";
    IdentTable.print();

    return 0;
}
```

Таблица строковых литералов

```
<INITIAL>(\")        {
                         BEGIN(STR);
                         yymore();
                     }

...

<STR>\"              {
                         EscapeStrLexeme();
                         BEGIN(INITIAL);
                         return TOKEN_CONST_STR;
                     }
```

```
void CoolLexer::EscapeStrLexeme()
{
    std::string str(yytext, yyleng);

    // remove leading and trailing '\"
    str = str.substr(1, str.length() - 2);
    std::string slex = "";
    std::string::size_type pos;


    while ((pos = str.find_first_of("\\")) != std::string::npos) {
        // ...
    }
    slex += str;

    // Add to symbol table (table of string literals)
    cool_yylval.symbol =
        StrTable.AddString(const_cast<char *>(slex.c_str()));
}
```

Таблицы идентификаторов и целочисленных литералов

```
f(?i:alse)          { AddBoolLexeme(false); return TOKEN_KW_FALSE; }
t(?i:rue)           { AddBoolLexeme(true); return TOKEN_KW_TRUE; }

 /* Integer decimal literal */
{num}               { AddIntLexeme(); return TOKEN_CONST_INT; }

 /* Identifires */
[A-Z][_a-zA-Z0-9]*       { AddIdentLexeme(); return TOKEN_IDENT_TYPE; }
[a-z][_a-zA-Z0-9]*       { AddIdentLexeme(); return TOKEN_IDENT_OBJ; }
[_a-zA-Z][_a-zA-Z0-9]*   { AddIdentLexeme(); return TOKEN_IDENT; }

%%

void CoolLexer::AddIntLexeme()
{
    cool_yylval.symbol = IntTable.AddString(const_cast<char *>(YYText()));
}

void CoolLexer::AddBoolLexeme(bool value)
{
    cool_yylval.boolean = value;
}

void CoolLexer::AddIdentLexeme()
{
    cool_yylval.symbol = IdentTable.AddString(const_cast<char *>(YYText()));
}

void CoolLexer::Error(const char *msg) const
{
    cool_yylval.error_msg = const_cast<char *>(msg);
}
```

Таблицы символов

```cpp
template <class Elem> class StringTable
{
protected:
    List<Elem> *table = nullptr;  // Linked list
    int index = 0;
public:
    Elem *AddString(char *str);
    Elem *AddString(char *str, int maxchars);
    Elem *AddInt(int val);
    Elem *LookupIndex(int index);
    Elem *LookupString(char *str);

    // Iterator
    int First() { return 0; }
    int More(int i) { return i < index; }
    int Next(int i) { return i + 1; }

    void print() { ListPrint(std::cerr, table); }
};

class CoolIdentTable : public StringTable<IdentEntry> { };
class CoolStringTable : public StringTable<StringEntry> { };
class CoolIntTable : public StringTable<IntEntry> { };

CoolIdentTable IdentTable;
CoolIntTable IntTable;
CoolStringTable StrTable;
```

Элемент таблицы символов (Entry)

```
class Entry;
using Symbol = Entry *;

class Entry {
protected:
    char *str = nullptr;
    int len = 0;
    int index = 0; // unique index for each string in the table
public:
    Entry(char *str, int len, int index)
    {
        this->str = new char [len + 1]; std::strncpy(this->str, str, len); this->str[len] = '\0';
    }
    char *GetString() const { return str; };
    int GetLen() const { return len; };
    bool IsEqualString(char *s, int len) const { return (this->len == len) && (std::strncmp(this->str, str, len) == 0); }
};

class StringEntry : public Entry {
public:
    StringEntry(char *str, int len, int index) : Entry(str, len, index) { };
};

class IdentEntry : public Entry {
public:
    IdentEntry(char *str, int len, int index) : Entry(str, len, index) { };
};

class IntEntry: public Entry {
public:
    IntEntry(char *str, int len, int index) : Entry(str, len, index) { };
};
```

Добавление элемента в таблицу символов

```
template <class T>
class List {
private:
    T *head;
    List<T> *tail;

public:
    List(T *head, List<T> *tail = nullptr): head(head), tail(tail) { }

    T *Head() const { return head; }

    List<T> *Tail() const { return tail; }
};


template <class Elem>
Elem *StringTable<Elem>::AddString(char *str, int maxchars)
{
    int len = std::strlen(str);
    len = std::min(len, maxchars);
    for (List<Elem> *list = table; list != nullptr; list = list->Tail()) {
        if (list->Head()->IsEqualString(str, len))
            return list->Head();
    }
    Elem *elem = new Elem(str, len, index++);
    table = new List<Elem>(elem, table);
    return elem;
}
```

Отладочная печать таблицы символов

```
-- test.cl
class Silly {
    copy() : SELF_TYPE { self };
};


class Sally inherits Silly { };


class Main {
    x : Sally <- (new Sally).copy();
    main() : Sally { x };
};
```

```
$ ./driver ./test.cl

Integer lexeme symbol table:

String lexeme symbol table:

Identifier symbol table:
x Sally main copy Sally Sally x Main Silly Sally self
SELF_TYPE copy Silly
```

Синтаксический анализатор

Описание грамматики языка Cool для Bison

Структуры данных для хранения AST

Грамматика

$$
\begin{array}{rcl}
\textit{program} & ::= & [\![\textit{class};]\!]^{+} \\
\textit{class} & ::= & \textbf{class}\ \text{TYPE}\ [\textbf{inherits}\ \text{TYPE}]\ \{\ [\![\textit{feature};]\!]^{*}\} \\
\textit{feature} & ::= & \text{ID}(\ [\ \textit{formal}\ [\![,\textit{formal}]\!]^{*}\ ]\ ) : \text{TYPE}\ \{\ \textit{expr}\ \} \\
 & | & \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ] \\
\textit{formal} & ::= & \text{ID} : \text{TYPE} \\
\textit{expr} & ::= & \text{ID}\ \texttt{<-}\ \textit{expr} \\
 & | & \textit{expr}[@\text{TYPE}].\text{ID}(\ [\ \textit{expr}\ [\![,\textit{expr}]\!]^{*}\ ]\ ) \\
 & | & \text{ID}(\ [\ \textit{expr}\ [\![,\textit{expr}]\!]^{*}\ ]\ ) \\
 & | & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{expr}\ \textbf{else}\ \textit{expr}\ \textbf{fi} \\
 & | & \textbf{while}\ \textit{expr}\ \textbf{loop}\ \textit{expr}\ \textbf{pool} \\
 & | & \{\ [\![\textit{expr};]\!]^{+}\} \\
 & | & \textbf{let}\ \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ]\ [\![,\text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ]]\!]^{*}\ \textbf{in}\ \textit{expr} \\
 & | & \textbf{case}\ \textit{expr}\ \textbf{of}\ [\![\text{ID} : \text{TYPE} => \textit{expr};]\!]^{+}\textbf{esac} \\
 & | & \textbf{new}\ \text{TYPE} \\
 & | & \textbf{isvoid}\ \textit{expr} \\
 & | & \textit{expr} + \textit{expr} \\
 & | & \textit{expr} - \textit{expr} \\
 & | & \textit{expr} * \textit{expr} \\
 & | & \textit{expr} / \textit{expr} \\
 & | & \sim\textit{expr} \\
 & | & \textit{expr} < \textit{expr} \\
 & | & \textit{expr} <= \textit{expr} \\
 & | & \textit{expr} = \textit{expr} \\
 & | & \textbf{not}\ \textit{expr} \\
 & | & (\textit{expr}) \\
 & | & \text{ID} \\
 & | & \text{integer} \\
 & | & \text{string} \\
 & | & \textbf{true} \\
 & | & \textbf{false}
\end{array}
$$

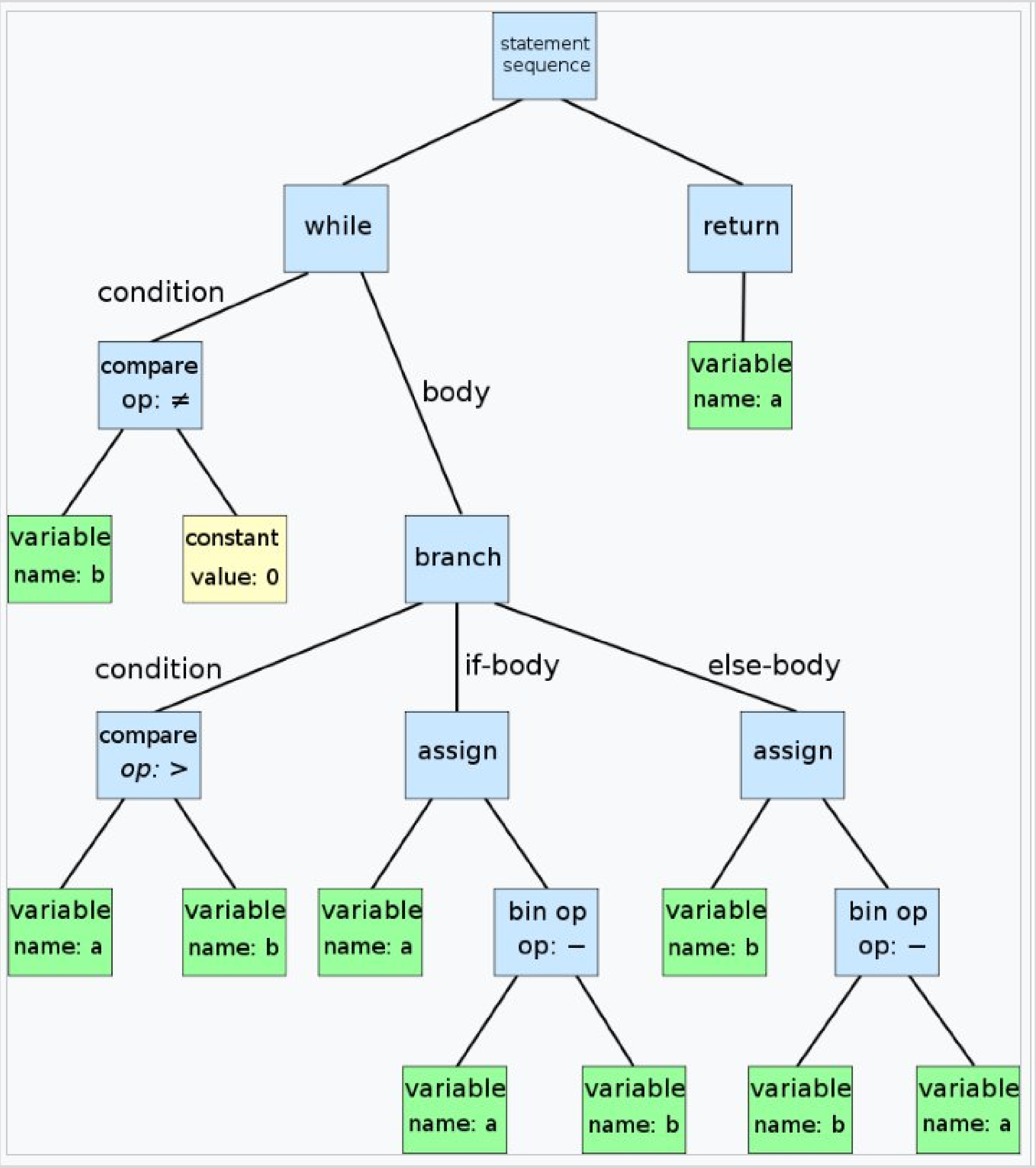
Precedence of infix binary and prefix unary operations:

```
.
@
~
isvoid
* /
+ -
<= < =
not
<-
```

- all binary operations are left-associative
- assignment is right-associative
- three comparison operations do not associate

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)

Описание грамматики

```
/* Parser definition for the Cool language */
%{
extern char *curr_filename;

/* Default locations represent a range in the source file, but this is not a requirement */
#define YYLTYPE int

/* The function yyparse expects to find the textual location of a token just parsed in the global variable yylloc. */
#define cool_yylloc curr_lineno

/*
 * YYLLOC_DEFAULT macro is invoked each time a rule is matched, before the associated action is run.
 */
extern int node_lineno;
#define YYLLOC_DEFAULT(Current, Rhs, N)  { Current = Rhs[1]; node_lineno = Current; }
#define SET_NODELOC(Current)  { node_lineno = Current; }

/* Root of AST */
Program ast_root;

/* List of classes */
Classes parse_results;

/* Number of parsing errors */
int omerrs = 0;

void yyerror(char *s);
extern int yylex();

%}
```

Описание грамматики

```
/*
 * The %union declaration specifies the entire collection of possible data types for semantic values.
 * The keyword %union is followed by braced code containing the same thing that goes inside a union in C.
 */
%union {
  Boolean boolean;
  Symbol symbol;
  Program program;
  Class_ class_;
  Classes classes;
  Feature feature;
  Features features;
  Formal formal;
  Formals formals;
  Case case_;
  Cases cases;
  Expression expression;
  Expressions expressions;
  char *error_msg;
}

/* Token kinds (terminals) */
%token CLASS 258 ELSE 259 FI 260 IF 261 IN 262
%token INHERITS 263 LET 264 LOOP 265 POOL 266 THEN 267 WHILE 268
%token CASE 269 ESAC 270 OF 271 DARROW 272 NEW 273 ISVOID 274
%token <symbol> STR_CONST 275 INT_CONST 276
%token <boolean> BOOL_CONST 277
%token <symbol> TYPEID 278 OBJECTID 279
```

Описание грамматики

```
/* Types of non-terminals */
%type <program> program
%type <classes> class_list
%type <class_> class


%type <features> feature_list
%type <feature> feature


%type <formals> formal_list
%type <formal> formal


%type <cases> case_list
%type <case_> case


%type <expressions> expr_list_simicolon expr_list_comma
%type <expression> expr


%type <expression> let_expr
%type <expression> optional_assign
%type <expression> let_binding_list
```

program	::=	⟦*class*;⟧$^+$
class	::=	**class** TYPE [**inherits** TYPE] { ⟦*feature*;⟧*}
feature	::=	ID([*formal* ⟦, *formal*⟧*]) : TYPE { *expr* }
	|	ID : TYPE [<- *expr*]
formal	::=	ID : TYPE
expr	::=	ID <- *expr*
	|	*expr*[@TYPE].ID([*expr* ⟦, *expr*⟧*])
	|	ID([*expr* ⟦, *expr*⟧*])
	|	**if** *expr* **then** *expr* **else** *expr* **fi**
	|	**while** *expr* **loop** *expr* **pool**
	|	{ ⟦*expr*;⟧$^+$}
	|	**let** ID : TYPE [<- *expr*] ⟦, ID : TYPE [<- *expr*]⟧* **in** *expr*
	|	**case** *expr* **of** ⟦ID : TYPE => *expr*;⟧$^+$**esac**
	|	**new** TYPE
	|	**isvoid** *expr*
	|	$expr + expr$
	|	$expr - expr$
	|	$expr * expr$
	|	$expr / expr$
	|	$\sim expr$
	|	$expr < expr$
	|	$expr <= expr$
	|	$expr = expr$
	|	**not** *expr*
	|	(*expr*)
	|	ID
	|	integer
	|	string
	|	**true**
	|	**false**

Описание грамматики

```
/* Precedence declarations */
%left IN
%right ASSIGN
%left NOT
%nonassoc LE '<' '='
%left '+' '-'
%left '*' '/'
%left ISVOID
%left '~'
%left '@'
%left '.'
```

```
Precedence of infix binary and prefix unary operations:
.
@
~
isvoid
* /
+ -
<= < =
not
<-

•  all binary operations are left-associative
•  assignment is right-associative
•  three comparison operations do not associate
```

program	::=	⟦*class*;⟧$^+$
class	::=	**class** TYPE ⟦**inherits** TYPE⟧ { ⟦*feature*;⟧*}
feature	::=	ID([*formal* ⟦, *formal*⟧*]) : TYPE { *expr* }
	|	ID : TYPE [<- *expr*]
formal	::=	ID : TYPE
expr	::=	ID <- *expr*
	|	*expr*⟦@TYPE⟧.ID([*expr* ⟦, *expr*⟧*])
	|	ID([*expr* ⟦, *expr*⟧*])
	|	**if** *expr* **then** *expr* **else** *expr* **fi**
	|	**while** *expr* **loop** *expr* **pool**
	|	{ ⟦*expr*;⟧$^+$}
	|	**let** ID : TYPE [<- *expr*] ⟦, ID : TYPE [<- *expr*]⟧* **in** *expr*
	|	**case** *expr* **of** ⟦ID : TYPE => *expr*;⟧$^+$**esac**
	|	**new** TYPE
	|	**isvoid** *expr*
	|	$expr + expr$
	|	$expr - expr$
	|	$expr * expr$
	|	$expr\, /\, expr$
	|	$\sim expr$
	|	$expr < expr$
	|	$expr <= expr$
	|	$expr = expr$
	|	**not** *expr*
	|	(*expr*)
	|	ID
	|	integer
	|	string
	|	**true**
	|	**false**

Описание грамматики

```
%%

/* Grammar rules */

program : class_list {  @$ = @1; ast_root = program($1); }
  /* @$ -- location of the whole grouping, @1 -- location of the first symbol */
;

class_list :
  class     /* single class */
  { $$ = single_Classes($1); parse_results = $$; }
| class_list class  /* several classes */
  { $$ = append_Classes($1, single_Classes($2)); parse_results = $$; }
| error ';' class
  { $$ = single_Classes($3); yyerrok; }
  /* macro yyerrok -- leave the error state before Bison finds the three good tokens */
;

/* Class inherits from the Object class */
class :
  CLASS TYPEID '{' feature_list '}' ';'
  { $$ = class_($2, idtable.add_string("Object"), $4, stringtable.add_string(curr_filename)); }
| CLASS TYPEID INHERITS TYPEID '{' feature_list '}' ';'
  { $$ = class_($2, $4, $6, stringtable.add_string(curr_filename)); }
;
```

Описание грамматики

```
/* Feature list (may be empty), but no empty features in list */
feature_list :
  /* empty */
  { $$ = nil_Features(); }
| feature_list feature /* multiple features */
  { $$ = append_Features($1, single_Features($2)); }
;

feature :
  OBJECTID ':' TYPEID optional_assign ';' /* attribute with initialize */
  { $$ = attr($1, $3, $4); }
| OBJECTID '(' formal_list ')' ':' TYPEID '{' expr '}' ';'
  { $$ = method($1, $3, $6, $8); }
| error ';' {}
;

formal_list :
  /* empty */
  { $$ = nil_Formals(); }
| formal /* single formal */
  { $$ = single_Formals($1); }
| formal_list ',' formal /* multiple formals */
  { $$ = append_Formals($1, single_Formals($3)); }
;

formal :
  OBJECTID ':' TYPEID
  { $$ = formal($1, $3); }
;
```

Описание грамматики

```
expr_list_comma :
  /* empty */
  { $$ = nil_Expressions(); }
| expr /* single expr */
  { $$ = single_Expressions($1); }
| expr_list_comma ',' expr
  { $$ = append_Expressions($1, single_Expressions($3)); }
;

expr_list_simicolon :
  expr ';' /* single expr */
  { $$ = single_Expressions($1); }
| expr_list_simicolon expr ';'
  { $$ = append_Expressions($1, single_Expressions($2)); }
| error ';' { yyerrok; }
;

expr :
  STR_CONST
  { $$ = string_const($1); }
| INT_CONST
  { $$ = int_const($1); }
| BOOL_CONST
  { $$ = bool_const($1); }
| OBJECTID
  { $$ = object($1); }
```

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathrm{ID} \texttt{ <- } \mathit{expr} \\
 & | & \mathit{expr}[@\mathrm{TYPE}].\mathrm{ID}(\ [\ \mathit{expr}\ \llbracket, \mathit{expr}\rrbracket^*\ ]\ ) \\
 & | & \mathrm{ID}(\boxed{[\ \mathit{expr}\ \llbracket, \mathit{expr}\rrbracket^*\ ]}) \\
 & | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{expr}\ \textbf{else}\ \mathit{expr}\ \textbf{fi} \\
 & | & \textbf{while}\ \mathit{expr}\ \textbf{loop}\ \mathit{expr}\ \textbf{pool} \\
 & | & \boxed{\{\ \llbracket \mathit{expr};\rrbracket^+ \}} \\
 & | & \textbf{let}\ \mathrm{ID} : \mathrm{TYPE}\ [\ \texttt{<-}\ \mathit{expr}\ ]\ \llbracket, \mathrm{ID} : \mathrm{TYPE}\ [\ \texttt{<-}\ \mathit{expr}\ ]\rrbracket^*\ \textbf{in}\ \mathit{expr} \\
 & | & \textbf{case}\ \mathit{expr}\ \textbf{of}\ \llbracket \mathrm{ID} : \mathrm{TYPE} \Rightarrow \mathit{expr};\rrbracket^+ \textbf{esac} \\
 & | & \textbf{new}\ \mathrm{TYPE} \\
 & | & \textbf{isvoid}\ \mathit{expr} \\
 & | & \mathit{expr} + \mathit{expr} \\
 & | & \mathit{expr} - \mathit{expr} \\
 & | & \mathit{expr} * \mathit{expr} \\
 & | & \mathit{expr} / \mathit{expr} \\
 & | & \sim\!\mathit{expr} \\
 & | & \mathit{expr} < \mathit{expr} \\
 & | & \mathit{expr} <= \mathit{expr} \\
 & | & \mathit{expr} = \mathit{expr} \\
 & | & \textbf{not}\ \mathit{expr} \\
 & | & (\mathit{expr}) \\
 & | & \mathrm{ID} \\
 & | & \mathrm{integer} \\
 & | & \mathrm{string} \\
 & | & \textbf{true} \\
 & | & \textbf{false}
\end{array}
$$

Описание грамматики

```
| expr '.' OBJECTID '(' expr_list_comma ')' /* dispatch */
  { $$ = dispatch($1, $3, $5); }
| OBJECTID '(' expr_list_comma ')'
  { $$ = dispatch(object(idtable.add_string("self")), $1, $3);
| expr '@' TYPEID '.' OBJECTID '(' expr_list_comma ')'
  { $$ = static_dispatch($1, $3, $5, $7); }

| IF expr THEN expr ELSE expr FI
  { $$ = cond($2, $4, $6); }
| WHILE expr LOOP expr POOL
  { $$ = loop($2, $4); }

| '{' expr_list_simicolon '}' /* blocks */
  { $$ = block($2); }

| let_expr
  { $$ = $1; }
| CASE expr OF case_list ESAC
  { $$ = typcase($2, $4); }

| NEW TYPEID
  { $$ = new_($2); }

| ISVOID expr
  { $$ = isvoid($2); }
```

$$
\begin{array}{rcl}
expr & ::= & \mathrm{ID} \texttt{ <- } expr \\
& | & expr[@\mathrm{TYPE}].\mathrm{ID}(\ [\ expr\ [\![, expr]\!]^*\ ]\ ) \\
& | & \mathrm{ID}(\ [\ expr\ [\![, expr]\!]^*\ ]\ ) \\
& | & \textbf{if } expr \textbf{ then } expr \textbf{ else } expr \textbf{ fi} \\
& | & \textbf{while } expr \textbf{ loop } expr \textbf{ pool} \\
& | & \{\ [\![expr;]\!]^+\} \\
& | & \textbf{let } \mathrm{ID} : \mathrm{TYPE}\ [\ \texttt{<-}\ expr\ ]\ [\![, \mathrm{ID} : \mathrm{TYPE}\ [\ \texttt{<-}\ expr\ ]]\!]^*\ \textbf{in } expr \\
& | & \textbf{case } expr \textbf{ of } [\![\mathrm{ID} : \mathrm{TYPE} => expr;]\!]^+ \textbf{esac} \\
& | & \textbf{new } \mathrm{TYPE} \\
& | & \textbf{isvoid } expr \\
& | & expr + expr \\
& | & expr - expr \\
& | & expr * expr \\
& | & expr\ /\ expr \\
& | & \sim expr \\
& | & expr < expr \\
& | & expr <= expr \\
& | & expr = expr \\
& | & \textbf{not } expr \\
& | & (expr) \\
& | & \mathrm{ID} \\
& | & \mathrm{integer} \\
& | & \mathrm{string} \\
& | & \textbf{true} \\
& | & \textbf{false}
\end{array}
$$

Описание грамматики

```
| expr '+' expr
  { $$ = plus($1, $3); }
| expr '-' expr
  { $$ = sub($1, $3); }
| expr '*' expr
  { $$ = mul($1, $3); }
| expr '/' expr
  { $$ = divide($1, $3); }
| '~' expr
  { $$ = neg($2); }
| expr '<' expr
  { $$ = lt($1, $3); }
| expr '=' expr
  { $$ = eq($1, $3); }
| expr LE expr
  { $$ = leq($1, $3); }
| NOT expr
  { $$ = comp($2); }

| '(' expr ')'
  { $$ = $2; }
;
```

Описание грамматики

```
case_list :
  /* empty */
  { $$ = nil_Cases(); }
| case_list case
  { $$ = append_Cases($1, single_Cases($2)); }
;

case :
  OBJECTID ':' TYPEID DARROW expr ';'
  { $$ = branch($1, $3, $5); }
;
```

Описание грамматики

```
let_expr :
  LET let_binding_list IN expr
  {
    auto bind = $2;
    bind->set_body($4);
    while (bind->parent) { bind = bind->parent; }
    $$ = bind;
  }
;

let_binding_list :
  let_binding
  { $$ = $1; }
| let_binding_list ',' let_binding
  { auto bind = $1; bind->set_body($3); $$ = $3; }
| error ',' let_binding { yyerrok; $$ = $3; }
;

let_binding :
  OBJECTID ':' TYPEID optional_assign
  { auto res = let($1, $3, $4, no_expr()); $$ = res; }
;

optional_assign :
  /* empty */
  { $$ = no_expr(); }
| ASSIGN expr
  { $$ = $2; }
;
```

Описание грамматики

```
%%

void yyerror(char *s)
{
    extern int curr_lineno;
    cerr << "\"" << curr_filename << "\", line " << curr_lineno << ": " \
         << s << " at or near ";
    print_cool_token(yychar);
    cerr << endl;
    omerrs++;

    if (omerrs > 10) {
        std::fprintf(stdout, "Error: more than 10 errors\n");
        std::exit(1);
    }
}
```

Построение абстрактного синтаксического дерева (AST)

```
class Main {
    main(): Int {
        10
    };
};
```

```
./myparser ./test.cl
#1
_program
  #1
  _class
    Main
    Object
    "./test.cl"
    (
    #2
    _method
      main
      Int
      #3
      _int
        10
      : _no_type
    )
```

```
program ::= [[class; ]]+
class   ::= class TYPE [inherits TYPE] { [[feature; ]]* }
feature ::= ID( [ formal [[, formal]]* ] ) : TYPE { expr }
          | ID : TYPE [ <- expr ]
formal  ::= ID : TYPE
expr    ::= ID <- expr
          | expr[@TYPE].ID( [ expr [[, expr]]* ] )
          | ID( [ expr [[, expr]]* ] )
          | if expr then expr else expr fi
          | while expr loop expr pool
          | { [[expr; ]]+ }
          | let ID : TYPE [ <- expr ] [[, ID : TYPE [ <- expr ]]]* in expr
          | case expr of [[ID : TYPE => expr; ]]+ esac
          | new TYPE
          | isvoid expr
          | expr + expr
          | expr − expr
          | expr ∗ expr
          | expr / expr
          | ~expr
          | expr < expr
          | expr <= expr
          | expr = expr
          | not expr
          | (expr)
          | ID
          | integer
          | string
          | true
          | false
```

Построение AST: литералы, идентификатор

cool.bison

```
%token <symbol> STR_CONST 275 INT_CONST 276
%token <boolean> BOOL_CONST 277
%token <symbol> TYPEID 278 OBJECTID 279
```

```
expr :
  STR_CONST
  { $$ = string_const($1); }
| INT_CONST
  { $$ = int_const($1); }
| BOOL_CONST
  { $$ = bool_const($1); }
| OBJECTID
  { $$ = object($1); }
```

ast.cpp

```
class string_const_class : public Expression_class {
protected:
   Symbol token;
public:
   string_const_class(Symbol s) { token = s; }
   void dump(ostream &stream, int n);
};

Expression string_const(Symbol token)
{
    return new string_const_class(token);
}

Expression int_const(Symbol token)
{
    return new int_const_class(token);
}

Expression object(Symbol name)
{
    return new object_class(name);
}
```

Построение AST: метод класса (feature)

cool.bison

```
feature :
  OBJECTID ':' TYPEID optional_assign ';'
  { $$ = attr($1, $3, $4); }

| OBJECTID '(' formal_list ')' ':' TYPEID '{' expr '}' ';'
  { $$ = method($1, $3, $6, $8); }
| error ';' {}
;
```

ast.cpp

```
class method_class : public Feature_class {
protected:
   Symbol name;
   Formals formals;
   Symbol ret_type;
   Expression expr;
public:
   method_class(Symbol a1, Formals a2, Symbol a3, Expression a4) : name(a1), formals(a2), ret_type(a3), expr(a4) { }
   void dump(ostream& stream, int n);
};

Feature method(Symbol name, Formals formals, Symbol return_type, Expression expr)
{
    return new method_class(name, formals, return_type, expr);
}
```

Построение AST: параметры метода

cool.bison

```
formal_list :
  /* empty */
  { $$ = nil_Formals(); }

| formal /* single formal */
  { $$ = single_Formals($1); }
| formal_list ',' formal /* multiple formals */
  { $$ = append_Formals($1, single_Formals($3)); }
;

formal :
  OBJECTID ':' TYPEID
  { $$ = formal($1, $3); }
;
```

ast.cpp

```
Formals nil_Formals()
{
   return new nil_node<Formal>();
}
```

Построение абстрактного синтаксического дерева (AST)

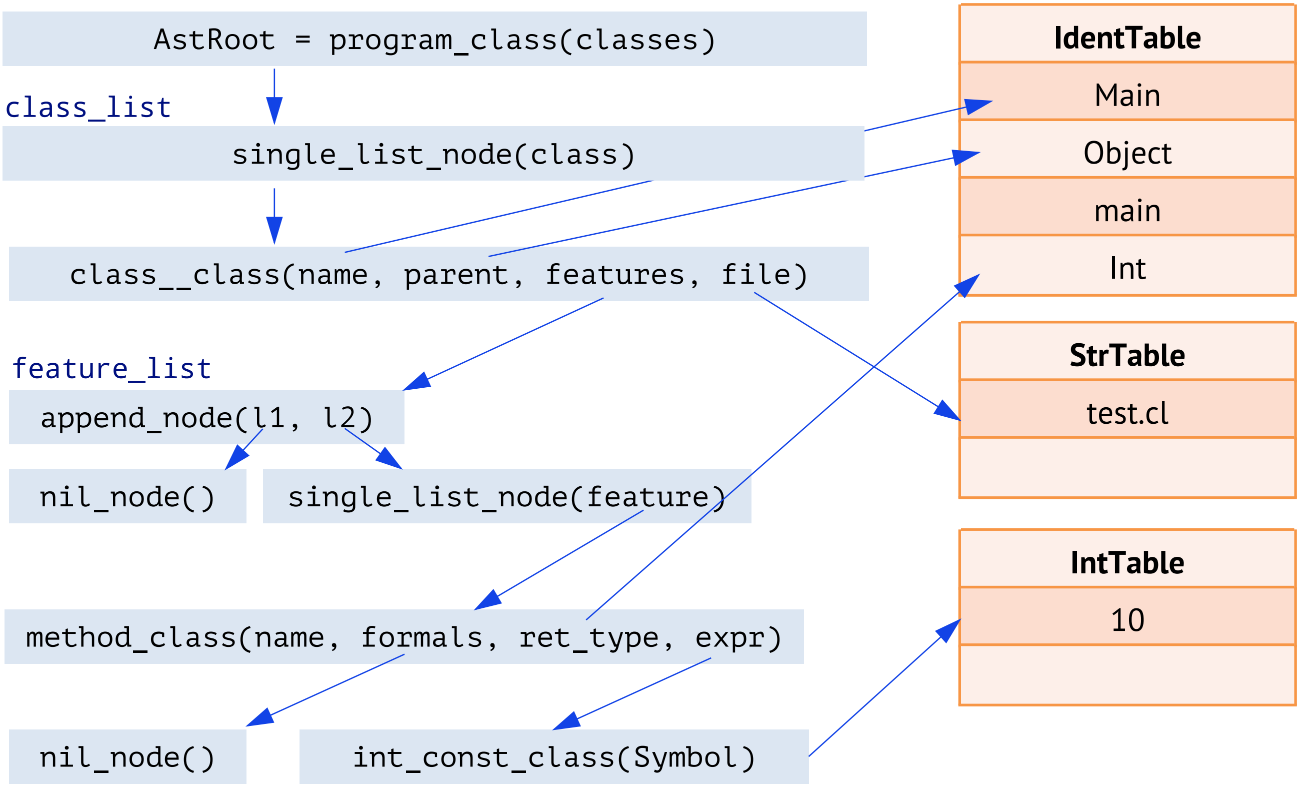
```
class Main {
    main(): Int {
        10
    };
};
```

```
SingleExpr = new IntConst(Symbol)

expr = new SingleListNode(SingleExpr)

formals = NilNode()

method = new Method(name, formals, return_type, expr)

features = new AppendListNode(new NilNode(),
                              new SingleFeatureList(method))

class = new Class(name, «Object», features, filename)

classes = new SingleListNode(class)

AstRoot = new program(classes)

AstRoot.dump() // visit all nodes
```

$$
\begin{array}{rcl}
\textit{program} & ::= & [\![\textit{class};]\!]^+ \\
\textit{class} & ::= & \boxed{\textbf{class}\ \text{TYPE}}\ [\textbf{inherits}\ \text{TYPE}]\ \{\ [\![\textit{feature};]\!]^*\} \\
\textit{feature} & ::= & \boxed{\text{ID}(\ [\ \textit{formal}\ [\![,\textit{formal}]\!]^*\ ]\ ) : \text{TYPE}\ \{\ \textit{expr}\ \}} \\
& | & \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ] \\
\textit{formal} & ::= & \text{ID} : \text{TYPE} \\
\textit{expr} & ::= & \text{ID}\ \texttt{<-}\ \textit{expr} \\
& | & \textit{expr}[@\text{TYPE}].\text{ID}(\ [\ \textit{expr}\ [\![,\textit{expr}]\!]^*\ ]\ ) \\
& | & \text{ID}(\ [\ \textit{expr}\ [\![,\textit{expr}]\!]^*\ ]\ ) \\
& | & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{expr}\ \textbf{else}\ \textit{expr}\ \textbf{fi} \\
& | & \textbf{while}\ \textit{expr}\ \textbf{loop}\ \textit{expr}\ \textbf{pool} \\
& | & \{\ [\![\textit{expr};]\!]^+\} \\
& | & \textbf{let}\ \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ]\ [\![,\text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \textit{expr}\ ]]\!]^*\ \textbf{in}\ \textit{expr} \\
& | & \textbf{case}\ \textit{expr}\ \textbf{of}\ [\![\text{ID} : \text{TYPE} \Rightarrow \textit{expr};]\!]^+\textbf{esac} \\
& | & \textbf{new}\ \text{TYPE} \\
& | & \textbf{isvoid}\ \textit{expr} \\
& | & \textit{expr} + \textit{expr} \\
& | & \textit{expr} - \textit{expr} \\
& | & \textit{expr} * \textit{expr} \\
& | & \textit{expr} / \textit{expr} \\
& | & \sim\textit{expr} \\
& | & \textit{expr} < \textit{expr} \\
& | & \textit{expr} <= \textit{expr} \\
& | & \textit{expr} = \textit{expr} \\
& | & \textbf{not}\ \textit{expr} \\
& | & (\textit{expr}) \\
& | & \text{ID} \\
& | & \boxed{\text{integer}} \\
& | & \text{string} \\
& | & \textbf{true} \\
& | & \textbf{false}
\end{array}
$$

Построение абстрактного синтаксического дерева (AST)

```
test.cl
class Main {
    main(): Int {
        10
    };
};
```




```
SingleExpr = new IntConst(Symbol)

expr = new SingleListNode(SingleExpr)

formals = NilNode()

method = new Method(name, formals, ret_type, expr)

features = new AppendListNode(new NilNode(),
                              new
SingleFeatureList(method))

class = new Class(name, «Object», features, filename)

classes = new SingleListNode(class)

AstRoot = new program(classes)

AstRoot.dump() // visit all nodes
```