Параллельная реализация языка программирования Рефал+: компиляция в императивный язык

Институт программных систем Российской академии наук

24 апреля 2003 г.

Аннотация

В отчете рассматриваются алгоритмы и формализмы, реализованные при разработке компилятора и run-time–библиотеки для конвертирования Рефал Плюс —программ в программы на императивном языке (на примере языка C++).

1 Схема компиляции Рефал Плюс в императивный язык

Общая схема компиляции программ на языке Рефал Плюс в императивный язык (на примере Си++) приведена на рис. 1 на стр. 1.

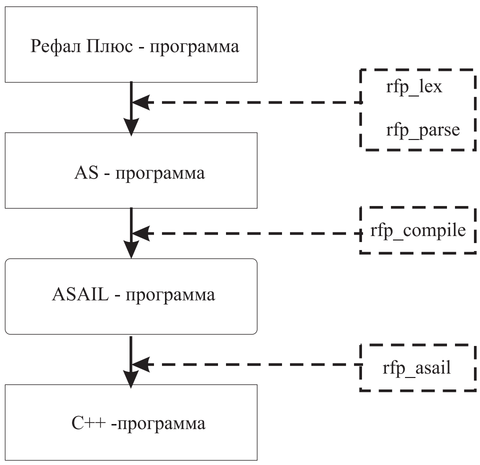

Рис. 1: Схема компиляции Рефала Плюс в императивный язык

В процессе компиляции выделяются три основных этапа:

1. Преобразование Рефал-программы в текст на языке абстрактного синтаксиса (AS-текст), которое выполняют модули `rfp_lex` и `rfp_parse`.

2. Преобразование AS-текста в программу на абстрактном императивном языке (ASAIL-текст), осуществляемое модулем `rfp_compile`.

3. Генерация выходного текста Си++ –программы, которую выполняет модуль `rfp_asail`.

Выбор такой многоступенчатой схемы компиляции программ на языке Рефал Плюс обусловлен следующими причинами:

1. Выделение AS-языка как входного языка для последующих этапов компиляции дает возможность использовать разработанные модули для компиляции в императивные языки других диалектов Рефала.

2. Использование в процессе компиляции абстрактного императивного языка позволяет решить задачу компиляции Рефала Плюс в любой другой императивный язык (в том числе Т-язык, используемый для динамического распараллеливания программ) путем замены блока, обеспечивающего генерацию выходного текста.

Более подробно основные этапы компиляции рассмотрены далее.

2 Этап компиляции "Рефал Плюс — Абстрактный синтаксис"

2.1 Правила преобразования в AS

При разработке абстрактного синтаксиса (далее AS) приоритетной была задача максимально обеспечить возможность отображения в AS-предложениях конструкций различных диалектов Рефала.

Вследствие этого был выбран следующий набор методов преобразования конструкций Рефала Плюс в AS-выражения:

1. Тело функции — путь (не обязательно блок).

2. Путь — линейная последовательность операторов, вытянутых через конкатенацию, а не через "cons".

3. Все AS-скобки, включая образующие термы, имеют впереди теги.

4. Сложные конструкции Рефала Плюс (перестройки, присваивания) разбиты на составляющие их действия.

5. Семантика действия состоит в пополнении среды и выработке нового форматного значения (объектного выражения со статическим форматом). Действия бывают простые (образцы, результаты, форматы) и сложные (блоки).

6. Есть отдельный оператор `NoFail`, непроницаемый для `$Fail`, и `CUTALL`, выражающий семантику равенства Рефал Плюс. Непрозрачному блоку в AS соответствует конструкция `NoFail (Block e.Branches)`.

7. Оператор итерации `$iter` заменяется более элементарными конструкциями `(Label sl)` и `(Loop sl)`. При этом в операторе `Loop` метка `sl` должна быть определена (оператором `Label`) выше на пути от начала функции к данному оператору `Loop`, т.е. ищется ближайшая точка `Label` с тем же символом-меткой `sl`.

8. Вместо трех понятий Рефала Плюс: `Tail`, `Path`, `Source`, — используется одно: `e.Sentence`.

9. Все константы подставлены на этапе лексического анализа. Ссылка на функцию и другие символы-ссылки выглядят как `(Ref s.Word)`. Определяющие вхождения задаются как `s.Word`. В вызове функции используется `(Ref s.Word)`.

10. Объектные символы (в отличие от символов-ссылок) представляются сами собой, а не как `(Symbol s)`.

11. Операторы `Error` и `Trap` имеют неявный дополнительный аргумент — продолжение справа от оператора.

12. Окончание выражений после операторов `Fail` и `Loop` игнорируется.

13. Конкретная реализация может не поддерживать какие-то элементы (или сочетания элементов) абстрактного синтаксиса. В этом случае должно выдаваться соответствующее сообщение.

2.2 Абстрактный синтаксис (AS)

```
t.Syntax           ::= t.Module | t.Interface
t.Module           ::= (MODULE t.ModuleName e.ModuleBody)
t.Interface        ::= (INTERFACE t.InterfaceName
                         e.InterfaceBody)
t.ModuleName       ::= t.Name
t.InterfaceName    ::= t.Name
e.ModuleBody       ::= t.Item | e.ModuleBody t.Item
t.Item             ::= t.Function | t.Object | t.Pragma
                       | t.Const
s.Linkage          ::= s.IntLinkage | s.ExtLinkage
s.IntLinkage       ::= EXPORT | LOCAL
s.ExtLinkage       ::= IMPORT
t.Function         ::= (s.IntLinkage s.FunctionTag
                         t.FunctionName t.InputFormat
                         t.OutputFormat e.Sentence)
                       |(s.ExtLinkage s.FunctionTag t.FunctionName
                         t.InputFormat t.OutputFormat)
s.FunctionTag      ::= FUNC | FUNC?
t.FunctionName     ::= t.Name
t.InputFormat      ::= t.Format
t.OutputFormat     ::= t.Format
t.Format           ::= (FORMAT e.HardExpression)
e.HardExpression   ::= e.HardTerms
```

```
                    | e.HardTerms t.EVar e.HardTerms
                    | e.HardTerms t.VVar e.HardTerms
e.HardTerms         ::= /*Empty*/ | t.HardTerm e.HardTerms
t.HardTerm          ::= t.Symbol | (PAREN e.HardExpression)
                    | t.SVar | t.TVar
t.Block             ::= (BLOCK e.Branches)
e.Branches          ::= /*Empty*/ | t.Branch e.Branches
t.Branch            ::= (BRANCH e.Sentence)
e.Sentence          ::= /*Empty*/ | t.Statement e.Sentence
t.Statement         ::= t.Action | t.Operator
                    | t.ComplexOperator | t.Pragma
t.Operator          ::= CUT | CUTALL | STAKE | NOFAIL
                    | FAIL | ERROR
t.ComplexOperator   ::= (NOT e.Sentence)
                    |(ITER t.IterBody t.IterVars t.IterCondition)
                    |(TRY t.TrySentence t.CatchSentence)
t.IterBody          ::= (e.Sentence)
t.IterVars          ::= t.Format
t.IterCondition     ::= (e.Sentence)
t.TrySentence       ::= (e.Sentence)
t.CatchSentence     ::= (e.Sentence)
t.Action            ::= t.Pattern | t.Result | t.Format | t.Block
t.Pattern           ::= (LEFT e.PatternExpression)
                    |(RIGHT e.PatternExpression)
e.PatternExpression ::= /*Empty*/
                    | t.PatternTerm  e.PatternExpression
t.PatternTerm       ::= t.Symbol | (PAREN e.PatternExpression)
                    | t.Variable
t.Variable          ::= t.EVar | t.VVar | t.TVar | t.Svar
t.EVar              ::= (EVAR t.Name)
t.VVar              ::= (VVAR t.Name)
t.TVar              ::= (TVAR t.Name)
t.SVar              ::= (SVAR t.Name)
t.Name              ::= (e.QualifiedName)
e.QualifiedName     ::= s.Word | e.QualifiedName  s.Word
t.Result            ::= (RESULT e.ResultExpression)
e.ResultExpression  ::= /*Empty*/
                    | t.ResultTerm e.ResultExpression
t.ResultTerm        ::= t.Symbol | t.Variable
                    |(PAREN e.ResultExpression)
                    |(CALL e.ResultExpression) | t.Block
t.Symbol            ::= s.ObjectSymbol | (REF t.Name)
s.ObjectSymbol      ::= s.Symbol
t.Object            ::= (s.Linkage s.ObjectType t.ObjectName)
s.ObjectType        ::= TABLE | BOX | STRING | VECTOR | CHANNEL
t.ObjectName        ::= t.Name
t.Initializer       ::= (EXEC e.ResultExpression)
t.Pragma            ::= (PRAGMA e.PragmaBody)
s.PragmaBody        ::= e.PragmaFile | e.PragmaLine
                    | e.PragmaTrace
```

```
e.PragmaFile        ::= FILE s.FileName
s.FileName          ::= s.Word
e.PragmaLine        ::= LINE s.Line s.Column
s.Line              ::= s.Number
s.Column            ::= s.Number
e.PragmaTrace       ::= TRACE e.TraceNames
e.TraceNames        ::= /*Empty*/ | t.TraceName e.TraceNames
t.TraceName         ::= t.Name
t.Const             ::=(s.ExtLinkage CONST t.ConstName)
                    |(s.IntLinkage CONST t.ConstName
                    e.ConstExpression)
t.ConstName         ::= t.Name
e.ConstExpression   ::= /*Empty*/
                    | t.ConstTerm e.ConstExpression
t.ConstTerm         ::= t.Symbol | (PAREN e.ConstExpression)
                    | t.Variable
s.Word              ::= символ-слово (в смысле Рефал Плюс)
s.Number            ::= символ-число (в смысле Рефал Плюс)
```

3 Этап компиляции "Абстрактный синтаксис (AS) — Абстрактный синтаксис императивного языка (ASAIL)"

3.1 Абстрактный синтаксис императивного языка

Набор конструкций абстрактного императивного языка (ASIL) включает элементы императивных языков (цикл, условный оператор, операторы перехода и др.), а также набор базовых функций run-time–библиотеки, ориентированных на работу с Рефал-выражениями.

Рассмотрим синтаксис абстрактного императивного языка, где в качестве комментариев (строки, начинающиеся с "//") даны пояснения смысла соответствующих конструкций.

```
e.ASAIL          ::= /*empty*/ | t.Func e.ASAIL
t.Func           ::= (FUNC t.QualifiedName (e.param)
                   (e.param) e.body)
// Определение  функции, где t.QualifiedName - имя функции,
// e.param, соответственно, входные и выходные параметры,
// e.body - тело функции.
t.QualifiedName ::= (e.name)
e.name           ::= /*empty*/ | s.numb e.name | s.word e.name
e.param          ::= /*empty*/ | t.var-ref e.param
t.var            ::=  t.var-ref | t.var-int
t.var-ref        ::= (VAR t.QualifiedName)
t.var-int        ::= (VAR t.QualifiedName)
e.body           ::= /*empty*/ | t.statement e.body
t.statement      ::= t.call | t.tailcall | t.assign | t.decl
                 |t.if | t.for | t.continue | t.break
                 |t.lsplit | t.rsplit | t.catch | t.try
```

```
                |t.drop | t.expr | t.deref | t.subexpr
                |t.constexpr | t.blok-label | t.decl-func
                |t.error
                | RETURN
// Оператор успешного выхода из функции.
                | FATAL
// Оператор аварии, вырабатывающий исключение
// вида Unexpected fail.
                | RETFAIL
// Оператор неуспешного выхода из функции (fail).
t.call          ::= (CALL t.QualifiedName (e.args) (e.args))
// Вызов функции, где t.QualifiedName - имя функции,
// e.args, соответственно, входные и выходные фактические
// параметры, передаваемые по значению.
e.args          :: = /*empty*/ | t.arg e.args
t.arg           ::= (VAR t.name) | (e.expr-ref) | (e.expr-int)
t.tailcall      ::= (TAILCALL t.QualifiedName (e.args) (e.args))
// "Хвостовой" вызов функции,
// где t.QualifiedName - имя функции,
// e.args, соответственно, входные и выходные
// фактические параметры, передаваемые по значению.
t.assign        ::= (ASSIGN t.var e.expr)
// Оператор присваивания, где t.var - имя переменной,
// e.expr - присваиваемое значение.
e.expr          ::=  e.expr-ref | e.expr-int
t.decl          ::= (DECL s.type t.var)
// Объявление переменной t.var типа s.type.
// Предполагается поддержка двух базовых типов:
// int (целые числа) и Expr (Рефал-выражения).
s.type          ::= "Expr" | "int"
t.if            ::= (IF (e.cond) e.body)
// Условный оператор, где e.cond - условие,
// e.body - последовательность действий, выполняемых
// в случае его истинности.
e.cond          ::= /*empty*/ | t.cond-term e.cond
// Операнды t.cond-term, входящие в условие e.cond,
// соединяются "И" (AND).
t.cond-term     ::= t.call
                |(SYMBOL? e.expr-ref (e.pos))
// Проверка структуры элемента, занимающего в
// выражении e.expr-ref позицию e.pos;
// условие истинно, если этот элемент - символ.
                |(EQ (e.expr-ref) (e.pos) (e.len)
                  (e.expr-ref) (e.pos) (e.len))
// Проверка равенства подвыражений
// выражений e.expr-ref1 и e.expr-ref2;
// e.pos1 и e.pos2 - начальные позиции сравниваемых
// подвыражений; e.len1 и e.len2, соответственно, их длины.
                |(FLAT-EQ (e.expr-ref) (e.pos) (e.expr-ref)
                  (e.pos) (e.len) )
```

```
// "Быстрая" проверка равенства подвыражений выражений
// e.expr-ref1 и e.expr-ref2;
// e.pos1 и e.pos2 - начальные позиции сравниваемых
// подвыражений; e.len - их длина;
// известно, что подвыражения не содержат скобочных термов.
               |(FLAT-SUBEXPR? e.expr-ref (e.pos) (e.len))
// Проверка структуры выражения e.expr-ref;
// условие истинно, если в  выражении e.expr-ref
// подвыражение, начинающееся в e.pos и имеющее длину e.len,
// не содержит скобочных термов.
               |(NOT e.cond)
// Логическое отрицание условия e.cond.
               |(INFIX s.op-bool e.expr-log)
               |(INFIX s.op-eq e.args-int)
e.pos          ::=e.expr-int
e.len          ::= e.expr-int
e.expr-log     ::= /*empty*/
               |t.cond-term e.expr-log
               |t.int-term e.expr-log
s.op-bool      ::= "&&" | "||"
s.op-eq        ::= "<" | ">" | "<=" | ">=" | "=="
t.for          ::= (FOR t.label (e.cond) (e.step) e.body)
// Цикл  FOR, где t.label - метка цикла
// (используется в операторе CONTINUE),
// e.cond - условие выполнения цикла,
// e.step  - оператор итерации,
// e.body - тело цикла.
t.label        ::= t.QualifiedName
e.step         ::= (INC-ITER t.var-ref)
// Шаг цикла FOR - инкремент итератора
// переменной t.var-ref, созданного ранее
// соответствующей функцией lsplit() или rsplit().
t.continue     ::= (CONTINUE t.label)
// Оператор перехода к следующей итерации цикла FOR,
// t.label - метка цикла.
t.break        ::= (BREAK t.label)
// Оператор выхода из цикла FOR,
// t.label - метка блока, содержащего цикл.
t.lsplit       ::= (LSPLIT e.expr-ref (e.min) t.left t.right)
t.rsplit       ::= (RSPLIT e.expr-ref (e.min) t.left t.right)
// Функции отщепления фрагментов Рефал-выражения
// (соответственно, слева и справа);
// e.expr-ref - исходное выражение;
// e.min - минимальная длина отщепляемого участка;
// t.left и t.right, соответственно, левая и правая части,
// на которые разделяется выражение.
t.left         ::= t.var-ref
t.right        ::= t.var-ref
e.min          ::= e.expr-int
t.try          ::= (TRY e.body)
```

```
// try-блок, где e.body - тело блока. Исключения из него
// перехватываются в соответствующем catch-блоке.
t.error         ::= (ERROR e.expr-ref)
// Оператор генерации исключения, где e.expr-ref - исключение.
t.catch         ::= (CATCH-ERROR e.body)
// Блок перехвата исключений, вырабатываемых соответствующим
// try-блоком. Для любого try-блока существует только один
// catch-блок, условие входа в который всегда  истинно
// (вида Expr const& err), анализ  и обработка исключения
// проводится в теле блока e.body.
t.drop          ::= (DROP t.var-ref)
// Блок, сигнализирующий о возможности освобождения
// памяти, занимаемой объектом  t.var-ref,
// т.к. она далее не используется.
t.expr          ::= (EXPR t.var-ref e.expr-ref)
// Конструктор объекта t.var-ref класса Expr
// со значением e.expr-ref.
t.deref         ::= (DEREF t.var-expr e.expr-ref (e.pos))
// Конструктор объекта t.var-ref класса Expr,
// значение которого - содержимое скобочного терма,
// занимающего в выражении e.expr-ref позицию e.pos.
t.subexpr       ::= (SUBEXPR t.var-ref e.expr-ref (e.pos)(e.len))
// Конструктор объекта t.var-ref класса Expr,
// который выделяет в выражении e.expr-ref,
// начиная с элемента e.pos, подвыражение длиной e.len.
e.pos           ::= e.expr-int
e.len           ::= e.expr-int
t.constexpr     ::= (CONSTEXPR t.QualifiedName e.const)
// Конструктор константного объекта класса Expr,
// где t.QualifiedName - имя константы, e.const - ее значение.
e.const         ::= /*empty*/ | s.symbol
                |(PAREN e.const) | (REF t.QualifiedName)
// Термы, входящие в e.const, соединяются операцией конкатенации.
t.blok-label    ::= (LABEL t.label  e.body)
// Помеченный блок, где e.body - тело блока, t.label - метка.
t.decl-func     ::= (DECL-FUNC  t.QualiiedName)
// Объявление функции с именем t.QualifiedName.
e.expr-ref      ::= /*empty*/ | t.expr-ref e.expr-ref
// Термы в Рефал-выражении объединяются конкатенацией.
t.expr-ref      :: = s.symbol | t.var-ref
                |(PAREN e.expr-ref)
// Оператор ''навешивания скобок'' для выражения e.expr-ref;
// результат типа Expr.
                |(DEREF e.expr-ref (e.pos))
// Извлечение содержимого скобочного терма,
// находящегося в e.expr-ref в позиции e.pos,
// результат типа Expr.
                |(SUBEXPR e.expr-ref (e.pos)(e.len))
// Выделение в выражении e.expr-ref подвыражения,
// начинающегося в e.pos и имеющего длину e.len;
```

```
// результат типа Expr.
               |(EXPR e.expr-value)
// Формирование выражения (e.expr-ref);
// результат типа Expr.
e.expr-value    ::= /*empty*/ | s.symbol e.expr-value
e.expr-int      ::= /* empty */ | t.int-term e.expr-int
// Термы в целочисленном выражении e.expr-int
// соединяются "+" (операция сложения); результат типа int.
t.int-term      ::= s.numb | t.var-int
                |(LENGTH e.expr-ref)
// Вычисление длины выражения e.expr-ref;
// результат типа int.
                |(MIN e.args-int) | (MAX e.args-int)
// Вычисление минимума (максимума) среди аргументов типа int;
// результат типа int.
                |(INFIX s.op-int e.args-int)
// Арифметические операции (левоассоциативные) для
// аргументов типа int.
e.args-int      ::= /*empty*/ | t.args-int e.args-int
t.args-int      ::= (e.expr-int)
s.op-int        ::= "+" | "-" | "*" | "/" | "%"
s.numb          ::= символ-число (в смысле Рефал Плюс)
s.word          ::= символ-слово (в смысле Рефал Плюс)
```

4 Этап компиляции "Абстрактный синтаксис императивного языка — Си++"

4.1 Основные типы данных

Основной тип данных — класс Expr, который описывает Рефал-выражение. Рассмотрим его структуру и интерфейс, который используются при генерации C++–кода:

```
class Expr {
private:
// указатель на первый терм
Term* first;
// указатель на следующий за последним термом
Term* last;
// указатель на владельца (holder) выражения в памяти
M_Chunk* mem_chunk;
// признак того, что в выражении имеются скобочные термы
uintptr_t flags;
public:
// конструктор от строки символов
inline Expr (char const* _s = null);
// конструктор от expression holder
inline Expr( Term const* _cp )
```

```
    // конструктор разыменования
inline Expr (Expr const& _expr, uintptr_t _pos);
    // конструктор подвыражения
inline Expr (Expr const& _expr, uintptr_t _pos, uintptr_t _length);
    // конструктор от скобочного терма (для выражения из одного терма)
inline Expr (Term const& _cp);
    // конструктор копирования
inline Expr (Expr const& _expr);
    // общие конструкторы
inline Expr (
Term* const _first, Term* const _last,
M_Chunk* const _mem_chunk, uintptr_t const _flags );
inline Expr (Expr const* _expr);
    // деструктор
inline ~Expr ();
    // удаление выражения (drop)
inline void drop();
    // функция печати выражения
inline void write (FILE* _fp) const;
    // функция печати с переводом строки
inline void writeln (FILE* _fp) const;
    // оператор присваивания
inline const Expr &operator=( const Expr &_expr );
    // конкатенация выражений
inline Expr operator + (Expr const& _expr) const;
    // конкатенация выражения с термом
inline Expr operator + (Term const& _term) const;
    // конкатенация терма с выражением
friend inline Expr operator + (Term const& _term, Expr const& _expr);
    // конкатенация термов
friend inline Expr operator + (Term const& _term1, Term const& _term2);
    // "навешивание скобок" на выражение
inline Term operator () () const;
    // "снятие скобок" со скобочного терма
inline Expr operator * () const;
    // возвращает true, если терм - символ
inline bool symbol_at (uintptr_t _pos) const;
    // возвращает true, если в выражении нет скобочных термов
inline bool is_flat () const;
    // возвращает true, если выражение - пусто
inline bool is_empty () const;
    // возвращает первый терм выражения
inline Term* get_first () const;
    // возвращает последний терм выражения
inline Term* get_last () const;
    // вычисление длины выражения
inline uintptr_t get_len () const;
    // проверка равенства выражений
friend inline bool eq (
Term const* _f1, Term const* _l1, Term const* _f2, Term const* _l2);
```

```
friend inline bool eq_expr (
const Expr &_e1, uintptr_t _pos1, uintptr_t _len1,
const Expr &_e2, uintptr_t _pos2, uintptr_t _len2 );
friend inline bool flat_eq (
const Expr &_e1, uintptr_t _pos1,
const Expr &_e2, uintptr_t _pos2, uintptr_t _len );
inline bool operator == (Expr const& _expr) const;
friend inline bool operator == (Term const& _t1, Term const& _t2);
    // проверка неравенства выражений
inline bool operator != (Expr const& _expr) const;
friend inline bool operator != (Term const& _t1, Term const& _t2);
}
```

4.2 Компиляция отщеплений

В данной реализации с компиляцией отщеплений фрагментов Рефал-выражений связаны два макроса:

- отщепление слева
  lsplit (expr, min, left, right)

- отщепление справа
  rsplit (expr, min, left, right),

где *expr* — входное выражение, *min* — длина отшепляемой части, *left* и *right* — выражения, получающиеся в результате отщепления, содержащие, соответственно, левую и правую части исходного выражения. В результате выполнения этих макросов образуются объекты класса SplitIterator, доступ к которым может быть осуществлен посредством макроса *iter(expr)*.

Ниже следует фрагмент C++–кода, содержащий отщепление, результаты которого используются в цикле:

```
lsplit(expr, min, expr1, expr2);
for (; ; iter(expr1)++) {
...
}
```

4.3 Генерация C++–текста

Основной тип объекта, над которым выполняются действия, — Рефал-выражение, для его описания в данной реализации определен класс Expr, рассмотренный ранее. Таким образом, для генерации выходного кода важно, что в ASAIL–программе используются три типа данных: целые (реализованный в C++ как int), Рефал-выражения (соответственно, класс Expr) и логические (как результат промежуточных вычислений в условных операторах).

В модуле генерации C++–программы выделяются правила, связанные с генерацией простых операторов и блоков (им соответствуют стандартные средства императивного языка) и генерацией специальных функций или методов для работы с объектами класса Expr. Также разделяются методы вычисления значений целочисленных, логических выражений и выражений типа Expr.

Рассмотрим правила соответствия конструкций абстрактного императивного языка выходным C++–конструкциям, использованные в данной реализации компилятора. В рассматриваемых примерах отражена общая схема преобразований, при этом преобразование ASAIL–выражения **X** обозначено конструкцией $[\![\mathbf{X}]\!]_{C++}$ и знаком "=".

A. Генерация операторов и блоков.

- **Определение функции**

$[\![\textit{(FUNC t.name (e.arg) (e.res) e.body)}]\!]_{C++} =$

RF_FUNC
($[\![\text{t.name}]\!]_{C++}$*, (RF_ARG* $[\![\text{e.arg}]\!]_{C++}$*), (RF_RES* $[\![\text{e.res}]\!]_{C++}$*))*
$[\![\textit{e.body}]\!]_{C++}$
RF_END

- **Вызовы функции**

$[\![\ \textit{(CALL t.name (e.args1) (e.args2))}]\!]_{C++} =$

RF_CALL ($[\![\text{t.name}]\!]_{C++}$*, (* $[\![\text{e.args1}]\!]_{C++}$*), (* $[\![\text{e.args2}]\!]_{C++}$*))*

$[\![\textit{(TAILCALL t.name (e.args1) (e.args2))}\ ]\!]_{C++} =$

RF_TAILCALL ($[\![\text{t.name}]\!]_{C++}$*, (* $[\![\text{e.args1}]\!]_{C++}$*), (* $[\![\text{e.args2}]\!]_{C++}$*))*

- **Оператор присваивания**

$[\![\textit{(ASSIGN t.var e.expr)}]\!]_{C++} =$

$[\![\textit{t.var}]\!]_{C++} = [\![\textit{e.expr}]\!]_{C++};$

- **Объявление переменной**

$[\![\textit{(DECL s.type t.var)}]\!]_{C++} =$

$[\![\textit{s.type}]\!]_{C++}\ \ [\![\textit{t.var}]\!]_{C++};$

- **Условный оператор**

$[\![\textit{(IF (e.cond) e.body)}]\!]_{C++} =$

if ($[\![\text{e.cond}]\!]_{C++}$*)*
{
    $[\![\text{e.body}]\!]_{C++}$
}

- **Генерация помеченного блока**

$[\![$(LABEL t.label e.body)$]\!]_{C++}$ =

```
{
    [[e.body]]C++
}
[[t.label]]C++: {}
```

- **Цикл FOR**

  $[\![$(FOR t.label (e.cond) (INC-ITER t.var) e.body)$]\!]_{C++}$ =

```
for ( ; [[e.cond]]C++ ; iter( [[t.var]]C++)++)
{
    [[e.body]]C++
    [[t.label]]C++ : { }
}
```

- **Оператор перехода к следующей итерации цикла for**

  $[\![$(CONTINUE t.label)$]\!]_{C++}$ =

  goto $[\![$t.label$]\!]_{C++}$;

- **Оператор выхода из цикла (t.label — метка блока, содержащего цикл)**

  $[\![$(BREAK t.label)$]\!]_{C++}$

  goto $[\![$t.label$]\!]_{C++}$;

- **Функция отщепления слева фрагмента Рефал-выражения**

  $[\![$LSPLIT e.expr (e.min) t.left t.right)$]\!]_{C++}$ =

  lsplit ($[\![$e.expr$]\!]_{C++}$, $[\![$e.min$]\!]_{C++}$, $[\![$t.left$]\!]_{C++}$, $[\![$t.right$]\!]_{C++}$);

- **Функция отщепления справа фрагмента Рефал-выражения**

  $[\![$(RSPLIT e.expr (e.min) t.left t.right)$]\!]_{C++}$ =

  rsplit ($[\![$e.expr$]\!]_{C++}$, $[\![$e.min$]\!]_{C++}$, $[\![$t.left$]\!]_{C++}$, $[\![$t.right$]\!]_{C++}$);

- **Оператор генерации исключения**

  $[\![$(ERROR e.expr)$]\!]_{C++}$ =

  error ($[\![$e.expr$]\!]_{C++}$);

- **try-блок**

  $[\![$(TRY e.body)$]\!]_{C++}$ =

trap
{
$[\![$*e.body*$]\!]_{C++}$
}

- **Блок перехвата исключений**

  $[\![$*(CATCH-ERROR e.body)*$]\!]_{C++} =$

  with
  {
  $[\![$*e.body*$]\!]_{C++}$
  }

- **Вызов метода освобождения памяти, занимаемой объектом t.var**

  $[\![$*(DROP t.var)*$]\!]_{C++} =$

  $[\![$*t.var*$]\!]_{C++}$*.drop ();*

- **Оператор успешного выхода из функции**

  $[\![$*RETURN*$]\!]_{C++} =$

  return true;

- **Оператор неуспешного выхода из функции**

  $[\![$*RETFAIL*$]\!]_{C++} =$

  retfail;

- **Оператор неуспешного выхода из функции**

  $[\![$*FATAL*$]\!]_{C++} =$

  error ("Unexpected fail");

- **Конструктор объекта класса Expr**

  $[\![$*(EXPR t.var e.expr)*$]\!]_{C++} =$

  Expr $[\![$*t.var*$]\!]_{C++}$ *(*$[\![$*e.expr*$]\!]_{C++}$*);*

- **Конструктор объекта класса Expr, значение которого — содержимое скобочного терма, занимающего в e.expr позицию e.pos**

  $[\![$*(DEREF t.var e.expr (e.pos))*$]\!]_{C++} =$

  Expr $[\![$*t.var*$]\!]_{C++}$ *(*$[\![$*e.expr*$]\!]_{C++}$*,* $[\![$*e.pos*$]\!]_{C++}$*);*

- **Конструктор объекта класса Expr, который является подвыражением выражения e.expr, начиная с элемента e.pos, длиной e.len**

  $[\![\mathit{(SUBEXPR\ t.var\ e.expr\ (e.pos)\ (e.len))}]\!]_{C++} =$

  $\mathit{Expr}\ [\![\mathit{t.var}]\!]_{C++}\ \mathit{(}[\![\mathit{e.expr}]\!]_{C++},\ [\![\mathit{e.pos}]\!]_{C++},\ [\![\mathit{e.len}]\!]_{C++}\mathit{);}$

- **Конструктор константного объекта класса Expr (термы, входящие в e.expr, объединены операцией конкатенации)**

  $[\![\mathit{(CONSTEXPR\ t.name\ e.expr\ )}]\!]_{C++} =$

  $\mathit{const\ Expr}\ [\![\mathit{t.name}]\!]_{C++} = [\![\mathit{e.expr}]\!]_{C++}\mathit{;}$

- **Объявление функции**

  $[\![\mathit{(DECL\_FUNC\ t.name)}]\!]_{C++} =$

  $\mathit{RF\_DECL\ (}[\![\mathit{t.name}]\!]_{C++}\mathit{);}$

B. Генерация логических операций, их результат — true или false.

- **Условие истинно, если элемент, занимающий в выражении e.expr позицию e.pos, является символом (в смысле Рефал+)**

  $[\![\mathit{(SYMBOL?\ e.expr\ (e.pos))}]\!]_{C++} =$

  $[\![\mathit{e.expr}]\!]_{C++}\mathit{.symbol\_at(}[\![\mathit{e.pos}]\!]_{C++}\mathit{)}$

- **Условие истинно, если равны подвыражения (e.expr1 и e.expr2 — исходные выражения; e.pos1 и e.pos2 — начальные позиции сравниваемых подвыражений, e.len1 и e.len2 — их длины)**

  $[\![\mathit{(EQ\ (e.expr1)(e.pos1)(e.len1)(e.expr2)(e.pos2)(e.len2))}]\!]_{C++} =$

  $\mathit{Expr :: eq\ (}[\![\mathit{e.expr1}]\!]_{C++},\ [\![\mathit{e.pos1}]\!]_{C++},\ [\![\mathit{e.len1}]\!]_{C++},$
  $[\![\mathit{e.expr2}]\!]_{C++},\ [\![\mathit{e.pos2}]\!]_{C++},\ [\![\mathit{e.len2}]\!]_{C++}\mathit{)}$

- **Условие истинно, если равны подвыражения (e.expr1 и e.expr2 — исходные выражения; e.pos1 и e.pos2 — начальные позиции сравниваемых подвыражений, e.len — их длина; известно, что подвыражения не содержат скобочных термов**

  $[\![\mathit{(FLAT\text{-}EQ\ (e.expr1)(e.pos1)(e.expr2)(e.pos2)(e.len))}]\!]_{C++} =$

  $\mathit{Expr :: flat\_eq}$
  $\mathit{(}[\![\mathit{e.expr1}]\!]_{C++},\ [\![\mathit{e.pos1}]\!]_{C++},\ [\![\mathit{e.expr2}]\!]_{C++},\ [\![\mathit{e.pos2}]\!]_{C++},\ [\![\mathit{e.len}]\!]_{C++}\mathit{)}$

- **Условие истинно, если в выражении e.expr подвыражение, начинающееся в e.pos и имеющее длину e.len, не содержит скобочных термов**

  $[\![$*(FLAT-SUBEXPR? e.expr (e.pos)(e.len))*$]\!]_{C++} =$

  $[\![$*e.expr*$]\!]_{C++}$*.flat_at(*$[\![$*e.pos*$]\!]_{C++}$*,* $[\![$*e.len*$]\!]_{C++}$*)*

- **Логическое отрицание**

  $[\![$*(NOT e.cond)*$]\!]_{C++} =$

  ! $[\![$*e.cond*$]\!]_{C++}$

- **Логическая операция (левоассоциативная)**

  $[\![$*(INFIX s.op (e.log1 e.log2 ...))*$]\!]_{C++} =$

  ($[\![$*e.log1*$]\!]_{C++}$ *s.op* $[\![$*e.log2*$]\!]_{C++}$ *s.op ...)*

C. Генерация методов работы с объектами класса Expr, их результат — объект класса Expr.

- **Операция "навешивания скобок"**

  $[\![$*(PAREN e.expr)*$]\!]_{C++} =$

  $[\![$*e.expr*$]\!]_{C++}$ *()*

- **Извлечение содержимого скобочного терма, находящегося в e.expr в позиции e.pos**

  $[\![$*(DEREF e.expr (e.pos))* $]\!]_{C++} =$

  Expr ($[\![$*e.expr*$]\!]_{C++}$*,* $[\![$*e.pos*$]\!]_{C++}$*)*

- **Выделение в выражении e.expr подвыражения, начинающегося в e.pos и имеющего длину e.len**

  $[\![$*(SUBEXPR e.expr (e.pos)(e.len))*$]\!]_{C++} =$

  Expr ($[\![$*e.expr*$]\!]_{C++}$*,* $[\![$*e.pos*$]\!]_{C++}$*,* $[\![$*e.len*$]\!]_{C++}$*)*

- **Создание объекта класса Expr**

  $[\![$*(EXPR e.expr)* $]\!]_{C++} =$

  Expr ($[\![$*e.expr*$]\!]_{C++}$*)*

D. Генерация целочисленных операций.

- **Вычисление длины выражения**

$[\![(LENGTH\ e.expr)]\!]_{C++} =$

$(int)\ [\![e.expr]\!]_{C++}.get\_len()$

- **Вычисление минимума из целых аргументов**

  $[\![(MIN\ e.args)]\!]_{C++} =$

  $pxx\_min\ ([\![e.args]\!]_{C++})$

- **Вычисление максимума из целых аргументов**

  $[\![(MAX\ e.args)]\!]_{C++} =$

  $pxx\_max\ ([\![e.args]\!]_{C++})$

- **Арифметическая операция (левоассоциативная)**

  $[\![(INFIX\ s.op\ (e.int1\ e.int2\ ...))]\!]_{C++} =$

  $([\![e.int1]\!]_{C++}\ s.op\ \ [\![e.int2]\!]_{C++}\ \ s.op...)$

5 Пример компиляции "Рефал Плюс — C++"

5.1 Компиляция специальных символов

В качестве ключевых слов в C++-программах используются следующие зарезервированные слова:

"asm"	"false"	"static"
"auto"	"float"	"static_cast"
"bool"	"for"	"struct"
"break"	"friend"	"switch"
"case"	"goto"	"template"
"catch"	"if"	"this"
"char"	"inline"	"throw"
"class"	"int"	"true"
"const"	"mutable"	"try"
"const_cast"	"namespace"	"typedef"
"continue"	"new"	"typeid"
"default"	"operator"	"typename"
"delete"	"private"	"union"
"do"	"protected"	"unsigned"
"double"	"public"	"using"
"dynamic_cast"	"register"	"virtual"
"long"	"reinterpret_cast"	"void"
"else"	"return"	"volatile"
"enum"	"short"	"wchar_t"
"explicit"	"signed"	"while"
"extern"	"sizeof"	

Для обозначения логических операций используются ключевые слова:

"and"	"and_eq"	"bitand"	"bitor"	"compl"
"not"	"not_eq"	"or"	"or_eq"	"xor"
"xor_eq"				

Специальные символы кодируются в соответствие с таблицей:

Рефал	C++	Рефал	C++	Рефал	C++
'!'	'e'	'?'	'q'	'_'	' '
'+'	'p'	'<'	'lt'	'>'	'gt'
'*'	'a'	'/'	's'	'␣'	'sp'
'|'	'bar'	'~'	'tilde'	'@'	'at'
'$'	'dollar'	'%'	'percnt'	'^'	'circ'
'('	'lpar'	')'	'rpar'	'['	'lsqb'
'{'	'lcub'	'}'	'rcub'	'`'	'grave'
'\"'	'quot'	'.'	'period'	','	'comma'
';'	'semi'	'-'	'm'	'='	'eq'
'\\'	'bs'	'#'	'num'	'&'	'amp'
']'	'rsqb'	'\´'	'apos'	':'	'colon'

Escape-последовательности кодируются в соответствие с Unicode–правилами:

Рефал-последовательность	C++–код
'\a'	'07'
'\b'	'08'
'\t'	'09'
'\n'	'0A'
'\v'	'0B'
'\f'	'0C'

5.2 Запуск компилятора

В состав компилятора входит ряд дополнительных модулей:

- rfpc — монитор, управляющий работотой всех модулей компилятора;
- rfp_as2as — модуль преобразования AS–текста с целью оптимизации;
- rfp_asail_optim — модуль оптимизации ASAIL–текста;
- rfp_format — модуль определения форматов выражений на этапе компиляции;
- rfp_check — модуль проверки корректности форматов;
- rfp_mangle — модуль конвертирования специальных символов;
- rfp_src — модуль настройки потоков ввода-вывода;
- rfp_err — модуль формирования сообщений об ошибках;

- rfp_helper — модуль некоторых часто используемых функций (abs, max и т.д.).

Команда запуска компилятора имеет формат:
rfpc [options] file1 [file2 ...],
где используются опции:

Опция	*Значение*
-I, -ipath <path>	Добавить путь <path> в список путей для поиска файлов интерфейса .rfi
-nc, -no-check	Отключить проверку синтаксиса.
-nt, -no-transform	Отключить преобразование AS-to-AS.
-v, -verbose	Выводить на экран информацию о ходе компиляции.
-u	Конвертировать символы-слова без учета регистра.
-o <name>	Направить вывод file1 в файл <name>.* или вывести в stdout.
-cc	Ввывести C++ – код.
-hh	Сгенерировать hh-файл.
-asr, -as-refal	Вывести текст в абстрактном синтаксисе (расширение .asr)
-ast, -as-transformed	Вывести текст в абстрактном синтаксисе после преобразования AS-to-AS (расширение .ast).
-asi, -as-ail	Вывести текст в абстрактном императивном языке (расширение .asi).
-no	Отключить оптимизацию asail-кода.

5.3 Пример компиляции Рефал Плюс–программы

В качестве иллюстрации к вышесказанному, рассмотрим результаты конвертирования Рефал Плюс–программы в C++–код.

Входной Рефал Плюс–текст:

```
$use StdIO;
Main = <PrintLN 'Hello!'>;
```

Выходной C++–текст:

```
#include <rf_core.hh>

namespace refal
{
using namespace rfrt;

namespace StdIO
{
```

```
RF_DECL(PrintLN);
}

namespace hello
{
RF_FUNC (Main, (), (RF_RES res_1))
Expr const_chars_H_e_l_l_o__ ("Hello!");
res_1 = empty;
RF_TAILCALL (StdIO::PrintLN, (const_chars_H_e_l_l_o__), ());
RF_END

rfrt::Entry rf_entry(Main);

}

}
```