**Московский государственный технический университет им. Н.Э. Баумана**

**Курсовой проект на тему:**

# «Условия РЕФАЛа-5 в Простом Рефале»

Выполнила:
Студентка группы ИУ9-72
Козлова Анастасия Александровна
Научный руководитель:
Коновалов Александр Владимирович

Москва, 2018г.

# Цель проекта

Целью данного курсового проекта является добавление в компилятор Простого Рефала на стадии анализа и синтеза эффективную поддержку условий.

# Условия Рефал-5

В общем виде предложение с условиями выглядит так:

**Pat, ResCond1 : PatCond1, ResCond2 : PatCond2, … = Res**

Условия в Рефал-5 записывается после образца и имеют вид:

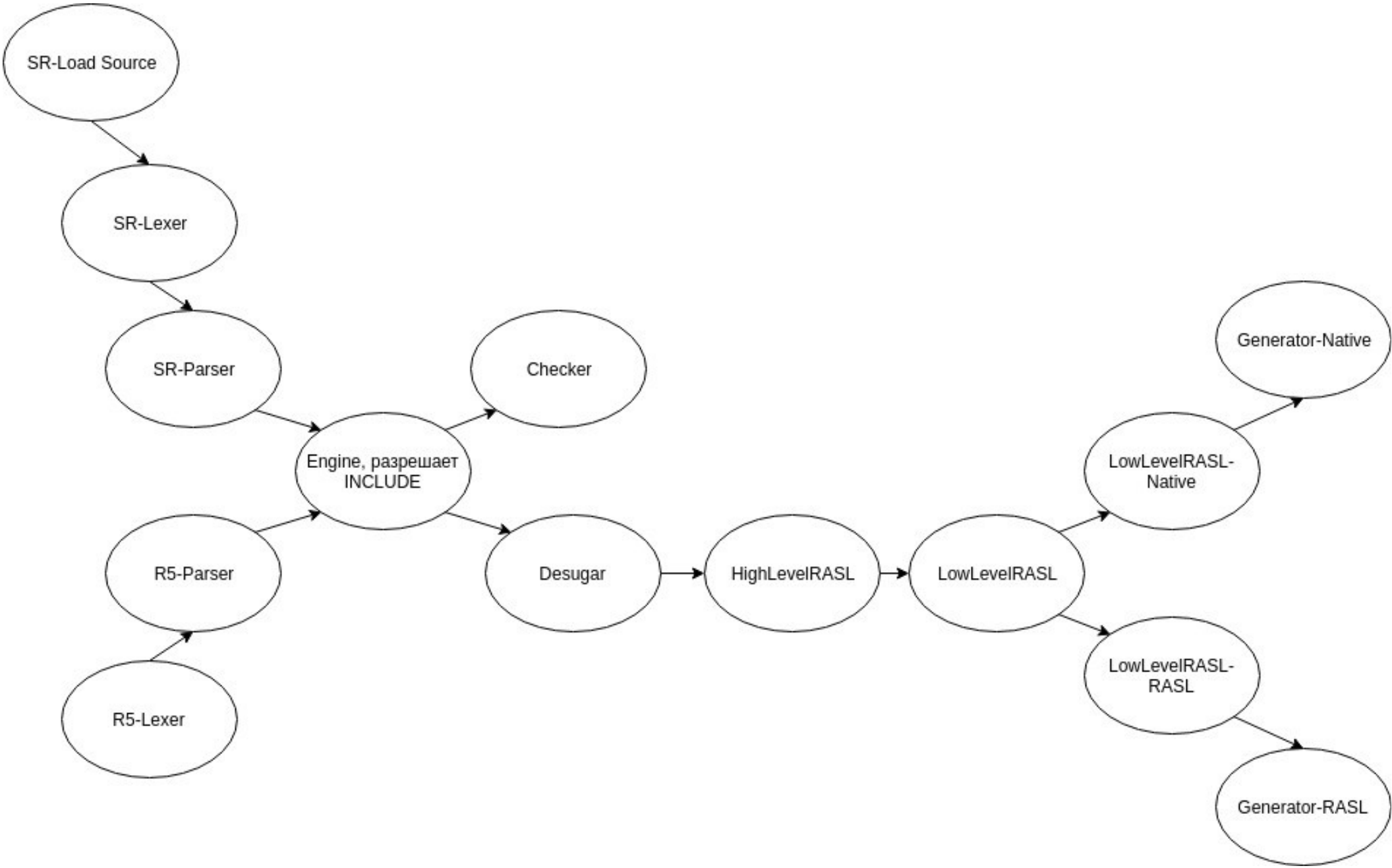
**, ResCond : PatCond**

, где

**ResCond** - произвольное результатное выражение, может содержать переменные, которые уже встречались в этом же предложении слева от запятой.

**PatCond** - произвольный образец, может содержать как старые, так и новые переменные

# Проходы компилятора



# Изменения на этапе синтаксического анализа

Для добавления поддержки условий потребовалось добавить в грамматику нетерминальный символ ConditionalPattern .

В итоге грамматика для предложения функции стала иметь вид:

```
Block = "{" {Sentence} "}".

Sentence = ConditionalPattern { Assign } "=" BlockedResult.

BlockedResult = Result { ":" Block }.

Assign = "=" BlockedResult ":" Pattern.

ConditionalPattern = Pattern { "," BlockedResult ":" Pattern }.
```

# Высокоуровневый RASL

Формат высокоуровневого RASLa для предложения функции F.

```
F {
 ...
 Pat , Res1 : Pat1, Res2 : Pat2 = Res;
 ...
}
```

```
(#CmdSentence
  команды сопоставления Pat
  команды порождения <F$k?1 Res1>
  выполнение подчиненного поля зрения
    (#CmdSentence
      команды сопоставления Pat1
      команды порождения <F$k?2 Res2>
      выполнение подчиненного поля зрения
        (#CmdSentence
          команды сопоставления с <F$k?1 Pat2>
          команды порождения Res
          (#CmdNextStep)
        )
        команда удаления <F$k?2 …>
      )
      команда удаления <F$k?1 …>
)
```

# Режим интерпретации

При итеративном выполнении условий нужно сохранять состояние Рефал-машины, а потом восстанавливать его. Для этого потребовалось создать структуру, описывающую состояние Рефал-машины.

Для восстановления и сохранения состояния Рефал-машины также потребовалось добавить 2 новые операции PopState и PushState.

Был добавлен тег #CmdCallCondition, который в режиме интерпретиции раскрывается как

**(#CmdCallCondition) =**

**(#CmdPushState)**

**(#CmdNextStep);**

```
1. struct StateRefalMachine{
2. refalrts::RefalFunction *callee;
3. refalrts::Iter begin; /* нужно для icSetResArgBegin в startup_rasl */
4. refalrts::Iter end;
5. const refalrts::RASLCommand *rasl;
6. refalrts::FunctionTableItem *functions;
7. const refalrts::RefalIdentifier *idents;
8. const refalrts::RefalNumber *numbers;
9. const refalrts::StringItem *strings;
10.
11. refalrts::vm::Stack<const refalrts::RASLCommand*> open_e_stack;
12. refalrts::vm::Stack<refalrts::Iter> context;
13.
14. refalrts::Iter res;
15. refalrts::Iter trash_prev;
16. int stack_top;
17.};
```

# Режим прямой кодогенерации

Для этого потребовалось, начиная с этапа генерации LowLevelRASL, разделить условия на #CmdConditionFunc-ToRasl и #CmdConditionFunc-ToNative.

Кроме того для рекурсивного вызова функции main_loop была создана функция recursive_call_main_loop:

```
refalrts::FnResult refalrts::recursive_call_main_loop() {
  const  refalrts::RASLCommand rasl[] = {
    { refalrts::icNextStep,0,0,0},
  };
  return refalrts::vm::main_loop(rasl);
}
```

Тег #CmdCallCondition в режиме прямой кодогенерации преобразовывается в следующий код:

```
 refalrts::FnResult rec_res =
refalrts::recursive_call_main_loop();
 if (rec_res != refalrts::cSuccess)
    return rec_res;
```

# Тестирование

Тестирование на генераторе случайных программ Рефала-5, автор которого - Немытых А.П.

Тестирование на Microsoft Windows 10.0 на компьютере с процессором Intel® Core™ i5-2430M CPU @ 2.40GHz, памятью 8 Гбайт при помощи компилятора Borland C++ 5.5.1

| Сравниваемые режимы | Время выполнения программы без оптимизации условий | Время выполнения программы с оптимизацией условий | Прирост производительности |
|---|---|---|---|
| -no opt / -OC | 10.603 | 10.250 | +3% |
| -OR/-ORC | 9.517 | 9.617 | -1% |
| -Od/-OdC | 9.039 | 8.671 | +4% |
| -OdR/-OdRC | 8.640 | 8.928 | -3% |

Тестирование на Ubuntu 16.04 на компьютере с процессором Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz, с памятью 8 Гбайт при помощи компилятора GCC 5.4.0.

| Сравниваемые режимы | Время выполнения программы без оптимизации условий | Время выполнения программы с оптимизацией условий | Прирост производительности |
|---|---|---|---|
| -no opt / -OC | 9.285 | 9.118 | +2% |
| -OR/-ORC | 7.982 | 8.188 | -3% |
| -Od/-OdC | 7.967 | 7.626 | +4% |
| -OdR/-OdRC | 6.702 | 6.794 | -1% |

# Тестирование

Парсер арифметических выражений, выполняющий разбор путём подбора возможных разбиений исходной строки. Тест написан специально для проверки условий.

Тестирование на Microsoft Windows 10.0 на компьютере с процессором Intel® Core™ i5-2430M CPU @ 2.40GHz, памятью 8 Гбайт при помощи компилятора Borland C++ 5.5.1

| Сравниваемые режимы | Время выполнения программы без оптимизации условий | Время выполнения программы с оптимизацией условий | Прирост производительности |
|---|---|---|---|
| -no opt / -OC | 8.979 | 6.330 | +30% |
| -OR/-ORC | 9.062 | 6.492 | +28% |
| -Od/-OdC | 8.457 | 6.041 | +29% |
| -OdR/-OdRC | 8.510 | 6.481 | +24% |

Тестирование на Microsoft Windows 10.0 на компьютере с процессором Intel® Core™ i5-2430M CPU @ 2.40GHz, памятью 8 Гбайт при помощи компилятора Borland C++ 5.5.1

| Сравниваемые режимы | Время выполнения программы без оптимизации условий | Время выполнения программы с оптимизацией условий | Прирост производительности |
|---|---|---|---|
| -no opt / -OC | 6.706 | 3.986 | +40% |
| -OR/-ORC | 5.028 | 4.218 | +16% |
| -Od/-OdC | 4.860 | 3.076 | +37% |
| -OdR/-OdRC | 3.033 | 2.938 | +3% |

# Заключение

В данной работе была полностью реализованная поддержка условий на всех проходах компиляции и во всех режимах компиляции. На основе результатов полученных при тестировании можно сделать вывод, что оптимизация с условиями не на всех программах дает значительный прирост производительности, а в случае, когда включены оптимизация с результатом и оптимизации с условием скорость работы программы может даже снижаться. Однако на программах, в которых в предложениях функций условия находятся в подавляющем количестве, увеличение производительности очень существенно и может достигать 30-40%.