Краткий tutorial для работы с кодом транслятора IndustrialC

1. Порядок работы, необходимый софт и background

Текущая версия транслятора сейчас лежит на https://github.com/deadproger/IndustrialC

Несмотря на то, что я использую студию, собирается все с помощью g++ из Mingw.

Для этого есть batch-скрипты: (я себе сделал кнопки в студии и привязал их к этим скриптам)

- build.bat - компилит и собирает IndustrialC.exe
- parse.bat - прогоняет препроцессор и транслятор для файла input.ic, генерит output.cpp
- compile.bat - компилит оттранслированный output.cpp с помощью avr-g++

Транслятор использует парсер-генератор Bison с лексическим анализатором Flex (Раньше они назывались Yacc и Lex соответственно). Лексический анализ (разбивка на токены) задан в tokens.l, парсинг происходит в parser.y. Там есть еще файлы prep_tokens.l и prep_parser.y - на них не надо обращать внимание, они сейчас никак не задействованы.

Соответственно, для работы нужно поставить MinGW, Bison и Flex. Подходит не всякая версия Bison/Flex. У меня стоит win_bison версии 3.0 и win_flex версии 2.5.37. Поставляется это в пакете “Win flex-bison” (лежит на SourceForge).

Также рекомендую ознакомиться с:

- общими принципами трансляции/компиляции и терминологией в этой области – синтаксическое дерево, терминальные/нетерминальные символы и т.п. (если на ФИТе не было предмета про это)
- bison и flex, в сети полно туториалов
- стандартами ANSI языка Си – там закреплена синтаксическая терминология (declaration, definition, statement, expression etc.), которая используется в трансляторе. В целом IndustrialC строился по принципу максимального соответствия синтаксису языка Си и тому, как он определяется в стандартах.

2. Общее описание

Трансляция производится в три шага:

- синтаксический анализ или парсинг (попутно идет лексический анализ)
- семантический анализ
- кодогенерация

Парсинг

Весь парсинг задается в файле parser.y. К сожалению, разделить этот файл нельзя - прикол Bison'a - поэтому файл большой. В этом файле задается грамматика языка в виде набора правил (как бэкус-науровы формы, но немного иначе). Для правил заданы куски С-кода, которые исполняются, когда это правило применяется. Нужно учитывать, что парсер обходит программу "в глубину", то есть он будет углубляться в дерево кода, пока не дойдет до листа (правила, где нет нетерминальных символов), выполнит код для этого правила, и тогда пойдет назад, вверх. То есть, например, код, ассоциированный с правилом для программы, выполнится последним, когда вся программа уже распарсена.

В результате парсинга строится AST (Abstract Syntax Tree). На самом деле тут это не совсем AST, но суть та же. Для каждой конструкции языка - программы, definition'ов, statemant'ов, expression'ов есть свой класс. Все эти классы унаследованы от iCNode - все они - узлы синтаксического дерева. При парсинге используется специальный объект ParserContext, который хранит в себе информацию, о том, где мы сейчас находимся - внутри процесса/состояния/функции, на какой строке какого файла мы сейчас и т .п.

Семантический анализ

При парсинге, те узлы дерева, для которых требуется провести дополнительный анализ, кладутся в список для второго прохода с помощью метода ParserContext::add_to_second_pass. Второй проход (семантический анализ) заключается в том, что для всех узлов из этого списка вызывается метод second_pass (см. main.cpp). Соответственно, эти узлы переопределяют у себя метод second_pass из iCNode и делают там, что им надо.

Кодогенерация

В iCNode есть абстрактный метод gen_code - он используется при кодогенерации, каждый класс-узел дерева должен его определить, то есть задать, во что он превратится в коде на Си. Эти методы обычно рекурсивные, то есть, например, процесс выведет switch{, потом вызовет gen_code у всех своих состояний, потом выведет }. При кодогенерации используется объект CodeGenContext, который, аналогично ParserContext хранит информацию о том, где мы сейчас находимся. Непосредственно генерация кода также осуществляется через методы CodeGenContext.

3. Экскурсия по коду

Чтобы примерно дать представление, как все устроено (и мне самому вспомнить, о чем важно сказать), я подробно описал свои действия при добавлении новой конструкции языка в транслятор. Это описание далеко не охватывает все особенности устройства транслятора, но должно изрядно помочь.

Предположим, я хочу добавить в язык конструкцию «restart;», которая перезапускает процесс. Семантически это эквивалентно «set state FS_START;». По сути это синтаксический сахар, который В.Е. недавно придумал для Рефлекса. Я не считаю, что он нужен в IndustrialC, но это хороший пример для данного случая – достаточно простой и в тоже время репрезентативный.

Ниже приведен порядок моих действий по добавлению этой новой, невероятной, волнующей и чрезвычайно полезной конструкции в язык. Критика, и осуждение с точки зрения людей, профессионально практикующих разработку софта, приветствуются.

1. Добавляем терминальный символ (токен) restart.

1.1. Добавляем TRESTART к списку токенов в parser.y. В угловых скобках тип токена, тип token означает, что этот токен будет представлен как int (это определено в tokens.l), то есть как числовой идентификатор. Здесь мы так делаем потому, что нам не нужно при кодогенерации иметь саму строку "restart", а только важно знать, что это за токен. TRESTART – это то, как мы решили обозвать этот токен. Самая правая часть "restart" это название токена, которое будет использоваться парсером при выдаче ошибок.

1.2. Добавляем токен в лексический анализатор: файл tokens.l. Лексический анализатор работает с регулярными выражениями, но в данном случае все просто – нам нужен буквально символ, представляющий собой слово restart. Макросы TOKEN и SAVE_TOKEN определены

parser.y:

```
%token <string> TLAND           "&&"
%token <string> TR_ASSGN        ">>="
%token <string> TL_ASSGN        "<<="
%token <string> TPLUS_ASSGN     "+="
%token <string> TMINUS_ASSGN    "-="
%token <string> TASTERISK_ASSGN     "*="
%token <string> TDIV_ASSGN      "/="
%token <string> TPERC_ASSGN     "%="
%token <string> TAND_ASSGN      "&="
%token <string> TXOR_ASSGN      "^="
%token <string> TOR_ASSGN       "|="
%token <token> TRESTART         "restart"

/*********************************************/
/*                  NODES                    */
/*********************************************/
%type <hyperprocess>        hp_definition
%type <process>             proc_def
%type <state_list>          proc_body
```

tokens.l:

в начале файла. SAVE_TOKEN сохраняет текст токена в строку, а TOKEN просто записывает int-идентификатор токена.

```
65  "for"                   return TOKEN(TFOR          );
66  "atomic"                return TOKEN(TATOMIC       );
67
68  "true"                  return TOKEN(TTRUE         );
69  "false"                 return TOKEN(TFALSE        );
70
71  "restart"               return TOKEN(TRESTART      );
72
73  "void"                  SAVE_TOKEN; return TVOID    ;
74  "char"                  SAVE_TOKEN; return TCHAR    ;
75  "int"                   SAVE_TOKEN; return TINT     ;
76  "short"                 SAVE_TOKEN; return TSHORT   ;
```

restart с точки зрения грамматики Си – это statement, поэтому обзываем все это iCRestartStatement и

2. Добавляем класс iCRestartStatement

- создаем файлы .h и .cpp для него. По возможности добавляем их в фильтр Statements в .sln. Файл .cpp нужно добавить в build.bat, чтобы он компилился и участвовал в сборке.

build.bat:

```
iCIterationStatement.cpp ^
iCAtomicBlock.cpp ^
iCJumpStatement.cpp ^
iCResetTimeoutStatement.cpp ^
iCCastExpression.cpp ^
iCRestartStatement.cpp
@echo ----------------------------
```

- я изначально скопировал в эти файлы все из аналогичного стэйтмента reset timeout.

- для рестарта нужно знать только в каком процессе мы находимся, поэтому конструктор в аргументах принимает только ссылку на контекст.

- содержимое метода gen_code я изначально закомментил. Пусть ничего не генерирует, парсинг будет работать, а кодогенерацию можно сделать в последнюю очередь.

3. Добавляем правило в грамматику в парсере

- В parser.y вверху добавляем заголовок iCRestartStatement.h

```
41      #include "iCAtomicBlock.h"
42      #include "iCJumpStatement.h"
43      #include "iCResetTimeoutStatement.h"
44      #include "iCRestartStatement.h"
45
46      #include <stdio.h>
47      #include <stdarg.h>
48      #include <typeinfo>
```

- ищем, где определено правило для statement. В конец этого правила добавляем вариант с restart. TRESTART это определенный нами токен restart, TSEMIC это точка с запятой. При распарсивании этого правила все, что мы делаем, это создаем новый объект iCRestartStatement. В конструктор передается контекст, из которого конструктор вытащит и сохранит для себя процесс, в котором мы находимся. $$ - это bisin'овское значение текущего узла AST. $1 и $2 здесь это соответственно TRESTART и TSEMIC. $1;$2; написано для того, чтобы bison не ругался (warning) на неиспользуемые символы.

parser.y:

```
        $1;//suppress unused value warning
    }
|   TRESET TTIMEOUT TSEMIC { $$ = new iCResetTimeoutStatement(*parser_conte
|   TRETURN expr TSEMIC { $$ = new iCJumpStatement("return",    $2,     *pa
|   TRETURN TSEMIC       { $$ = new iCJumpStatement("return",    NULL,   *pa
|   TBREAK TSEMIC        { $$ = new iCJumpStatement("break",     NULL,   *pa
|   TCONTINUE TSEMIC     { $$ = new iCJumpStatement("continue",  NULL,   *pa
|   TRESTART TSEMIC { $$ = new iCRestartStatement(*parser_context); $2;}
;
```

4. Пишем конструктор iCRestartStatement и проверяем что все парсится, ничего не падает и не течет

- Идем в конструктор iCRestartStatement. Здесь мы вытаскиваем из контекста текущий процесс, сохраняем его у себя, проверяем что он не NULL (иначе мы были не внутри процесса –

iCRestartStatement.cpp:

надо выдать ошибку), сохраняем у себя имя этого процесса и строку в коде, где наш statement находился.

```
44  iCRestartStatement::iCRestartStatement( const ParserCon
45  :   iCNode(context)
46  {
47      proc = context.get_process();
48      if(NULL == proc)
49          err_msg("\"restart\" used outside of process");
50      else
51          proc_name = proc->name;
52
53      line_num = context.line();
54  }
```

В целях контроля я на этом этапе добавил в конструктор вывод в cout сообщения, что найден restart, добавил "restart; " в код в input.ic в разных местах, собрал транслятор (build.bat), транслировал input.ic (parse.bat) и убедился, что парсер все нормально ест, что он выдает ошибку и не падает, если restart встречен вне процесса. А также проверил, что то, что мы написали не истекает памятью – в конце транслятор выдает сообщение pointers still allocated: 3. Это количество указателей, которые не sadeletелись на момент перед окончанием трансляции. По хорошему там должен быть ноль, но так выходит, что нет. Это зависит от компилятора, и я так однозначно и не смог определить, что это за три указателя, которые остаются аллоцированными. Главное, что когда вы что-то добавляете это число не должно увеличиваться.

5. Пишем кодогенерацию

Идем в метод iCRestartStatement::gen_code, и пишем кодогенератор:

```
17  void iCRestartStatement::gen_code(CodeGenContext& context)
18  {
19      if(NULL == proc)
20          return;
21
22      bool need_atomic_block = !context.in_ISR() && (proc->is_isr_driven() || proc->is_isr_referenced());
23
24      //add atomic block if in background loop
25      if(need_atomic_block)
26          context.atomic_header();
27
28      context.set_location(line_num, filename);
29      context.indent();
30      context.to_code_fmt("%s(%s, %s);", C_STRANS_MACRO, proc_name.c_str(), START_STATE_NAME);
31
32      context.to_code_fmt("\n");
33
34      //atomic block footer
35      if(need_atomic_block)
36          context.atomic_footer();
37  }
```

- в начале выясняем, надо ли окружить эту инструкцию atomic block'ом. Поскольку операция рестарта предполагает изменение состояния и обнуление его времени (а это long, то есть 4 инструкции на 8-битной архитектуре), будет не очень весело, если, скажем, какой-нибудь процесс в прерывании влезет посреди reset'a и начнет что-то у нас менять – вызовет start, stop, или спросит, активен ли наш процесс.

- далее, если решили, что надо – вставляем atomic block. Это запретит все прерывания.

- метод context.set_location либо разместит в коде line-marker с указанием файла и строки, либо, если файл прежний, и строка не сильно далекая, просто добавит символов новой строки. Это все нужно, чтобы Си-компилятор потом корректно выдавал сообщения об ошибках. line_num и filename – это свойства любого узла AST, они определены в iCNode, от которого все узлы дерева наследуются.

- context.indent вставляет табы с учетом текущей глубины индентации, чтобы Си-код потом был читаемым

- to_code_fmt работает как print в Си, с форматными строками. Есть еще метод to_code_string, он ест сиплюсплюсный string, и метод to_code, который не учитывает символы новой строки в том коде, что мы ему скармливаем (костыль, который понадобился для цикла for).

- При помощи to_code_fmt мы вставляем макрос для set_state с аргументами – идентификатор процесса и идентификатор начального состояния. Эти строки прописаны в common.h. Изначально это была попытка организовать шаблоны кодогенерации, но она успешно заглохла.

- в конце добавляем новую строку, и, если был atomic block, его нужно закрыть.

После этого собираем транслятор, проверяем, что:

- транслятор собирается
- транслятор не падает
- память не течет – значение pointers still allocated осталось прежним
- в выходном коде на Си действительно выдается то, что мы хотели