spirit x3

A newest version claims fast in compilation itself.

첫 튜토리얼

https://ciere.com/cppnow15/x3_docs/spirit/tutorials/

가장 단순한 파서들로 시작해서 점점 더 어려운 개념들을 보여준다. calculator는 포함되어 있고 json 파서는 있다고 하니 찾아본다. 익숙해질 때까지 계속 연습한다.

Roman Numerals

https://ciere.com/cppnow15/x3_docs/spirit/tutorials/roman_numerals.html

앞 쪽은 PEG나 EBNF를 알면 쉽게 따라갈 수 있다. 여기서부터 제대로 된 튜토리얼이다. 큰 파서를 작성하려면 이 쪽을 알아야 한다.

Rule이 중요하다. PEG를 갖는다.

```
rule<ID, Attribute> const r = "some-name";       // ID can be any struct / class

auto const r_def = double_ >> *(',' >> double_); // _def attached to rule definition

BOOST_SPIRIT_DEFINE(r); // combines rule and rule definition
```

X3에서 문법은 룰들의 논리적인 그룹이다.

boost visualizers

https://marketplace.visualstudio.com/items?itemName=ArkadyShapkin.CDebuggerVisualizersforVS2017

AST

중요한 예제이고 마지막 예제이다. 이제 PEG를 정의하고 파싱부터 진행해서 AST를 만든다. AST에 기반해서 C++ 코드를 생성한다. 주석을 포함할 수 있도록 한다. C에 가까운 PEG를 사용한다.

https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm

C 문법의 EBNF 정의 항목이다. 이에 기초해서 하나씩 파서를 만들어 간다.

에러 처리를 포함하는 버전이 있으니 해당 부분을 참고해서 에러 처리 루틴을 작성하면 된다.

Calculator

CppCon에서 일부 발표된 부분이나 튜토리얼에는 설명이 없다. 파싱된 내용을 실행하는 부분이므로 AST와 의미 해석과 관련된 샘플로 살필 가치가 있다.

Tutorial

Document:

https://ciere.com/cppnow15/using_x3.pdf

Movie:

https://www.youtube.com/watch?v=xSBWklPLRvw&index=28&list=PLHTh1InhhwT75gykhs7pqcR_uSiG601oh

- DSEL
  - Expression Templates
- PEG
  - Parsing Expression Grammar
    - CFG (Context Free Grammar)와 유사하지만 모호함을 제거하고 더 간결하게 만들어짐
    - 파서를 더 간단하게 구현하려는 노력의 산물
  - +, *, |, >>
- Parsers
  - Float ...
- Synthesized Attribute

Parsers

int_ for integer parser (double_, float_)

lit("foo") : a literal parser

```
std::string input("1234");
x3::parse( input.cbegin(), input.cend(), int_);
```

Available Parsers

short, *int*, long_, long_long, int_,

bin, oct, hex, ushort_, ulong_, uint_, ulong_long, uint_,

float_, double_, long_double, double_

bool_, true_, false_

byte_, word, dword, qwrod, word

big_word, big_dword, big_qword, big_dword

little_word, little_dword, little_qword, little_qword

char_, char_('x'), char_(x), char_('a', 'z'), char_("a-z8AZ"), ~char_('a'), lit('a'), 'a'

string("foo"), string(s), lit("bar"), "bar", lit(s)

alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, xdigit

Combining Parsers

```
std::string input("876 1234.56");
x3::parse( input.cbegin(), input.cend(), int_ >> space >> double_ );
```

Operators:

- Sequence> a b : a >> b
- Alternative> a | b : a | b
- Kleene> a* : *a
- Plus> a+ : +a
- Optional> a? : -a
- And> &a : &a
- Not> !a : !a
- Difference> a - b
- Expectioan> a > b
- List> a % b

PEG: https://en.wikipedia.org/wiki/Parsing_expression_grammar

- And-predicate
  - 입력을 사용하지 않으면서 식 e가 성공하면 성공, 실패하면 실패
- Not
  - And-predicate의 역 (실패하면 성공, 성공하면 실패)

Rules

- allows to name parsers
- specify the attribute type
- allows for recursion
- error handling
- attach handlers when a match is found

```
auto name = alpha >> *alnum;
auto quote = '"' >> *( ~char_('"') ) >> '"';

auto name = x3::rule<class name>{} = alpha >> *alnum; // same
```

A complete example:

```
std::string input( "foo : bar , "
                    "gorp : smart , "
                   "falcou : \"crazy frenchman\" , "
                   "name : sam "
                  );

auto iter = input.begin();
auto iter_end = input.end();
auto name = alpha >> *alnum;
auto quote = '"'
>> lexeme[ *(~char_('"')) ]
>> '"';

phrase_parse(
    iter, iter_end,
    ( name >> ':' >> (quote | name) ) % ','
    , space
);
```

name / value pair를 가져온다.

x3에서는 grammar가 필요 없다. parser로 연결해서 바로 파싱이 가능.

Synthesized Attributes

parser들은 attribute를 노출시킨다.

예제 1:

```
std::string input( "1234" );
int result;
parse(
    input.begin(), input.end(),
    int_,
    result
);
```

Compatibility:

- Attribute parsing is where the Spirit Magic lives.

```
std::string input(
    "foo : bar ,"
    "gorp : smart ,"
    "falcou : \"crazy frenchman\" "
);

auto iter = input.begin();
auto iter_end = input.end();

auto name = rule<class name, std::string>()
         = alpha >> *alnum;
```

```
auto quote = rule<class quote, std::string>()
           = '"'
               >> lexeme[ *(~char_('"')) ]
              >> '"';

auto item = rule<class item,
             std::pair<std::string, std::string>>()
             name >> ':' >> ( quote | name );

std::map< std::string, std::string > key_value_map;
phrase_parse(
    iter, iter_end,
    item % ',',
    space,
    key_value_map
);
```

Rule의 (synthesized) attribute는 RHS (오른쪽) 정의와 호환되어야 한다.

위 샘플을 빌드하는 중 템플릿 빌드 관련 오류가 발생한다.

1> d:\laxtools\lax\ext\boost_1_66_0\boost\spirit\home\x3\support\traits\move_to.hpp(62): error C2679: 이항 '=': 오른쪽 피연산자로 'std::basic_string<char,std::char_traits,std::allocator>' 형식을 사용하는 연산자가 없거나 허용되는 변환이 없습니다.

찾기가 좀 까다롭기는 한데 x3가 fusion에 많이 의존하고 있어 다음을 포함해야 한다.

```
#include <boost/fusion/adapted/std_pair.hpp>
```

Tidbits

- Start small
- Comose and test
- Test early and often
- Parsing first, Attributes second
- Allow the natual AST to fall out
- Refine grammar/AST
- Avoid semantic actions! Generato ASTs instead
    - imperative semantic actions are ugly warts in an elegant declarative grammar
    - use semantic actions only to facilitate the generation of an attribute
    - if you really can't avoid semantic actions, at least make them side-effect free.
        - backtracking can cause havoc when actions are called multiple times.

예제로 함수 호출이 있는 calculator가 있다. 설명되지 않은 내용들이 많아서 동영상을 보면서 정리한다.

Exercise

include path;

namespace name.name;

enum id {

value [= default];

}[;]

struct id {

type value[=default];

}[;]

message id {

type value[=default];

type value[integer | enum];

}[;]

table id {

type value [=default | : attributes];

}[;]

attrubutes = primary | foreign full_table_name | unique

https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm

C 문법의 EBNF 정의 항목이다. 이에 기초해서 하나씩 파서를 만들어 간다.

include "path/to/a.idl"

- include literal
- path
  - validate generation
  - use pre-generated a.ast file if it exists
    - written in json

identifier

- letter -(*(letter | digit))
- letter <- alpha | _

- string

namespace

- identifier % '.'
- vector

enum

- "enum" >> identifier >> '{'
- enum_value
- '}' -';'
- enum_value <- identifier >> value_expression

struct

- "struct" >> identifier >> '{'
- type_decl % ';'
- '}'

type_decl

- type identifier
- type <- primitive_types (from symbol with string values) | type_identifier
- pair
- optional namespace

type_identifier

- -namespace >> identifier
- pair

variable

- identifier

default value

- = value_expression

array

- [value_expression]

message

- same as struct
- code generation is different

c++ / c#

- untouched section
- put it inside class
- literal >> '{'
- 	lexeme[문자열]
- '}'

주석

- input_line
- lexer 차원에서 처리
- skipper 로 처리
  - http://www.boost.org/doc/libs/1_57_0/libs/spirit/example/qi/compiler_tutorial/mini_c/skipper.hpp
  - 유사한 방식으로 Skipper 작성이 x3에서 가능할 듯

코드 생성

c#

- little endian
  - host가 big edian일 경우만 변환 (대부분의 경우 변환 없음)
  - 서버에서 리틀 엔디언으로 송수신
- Pack / Unpack 을 Stream에 대해 하는 함수를 추가

c++

- 함수 생성
  - 헤더 파일에만 정의
  - field.h
    - 메세지 헤더 정의
    - 함수 정의 추가
    - 기본 Set 함수 추가
  - field_gen.inl
    - 기본 함수 골격들 생성
    - 사용하는 건 선택할 일
    - 복사 후 구현 방식 추천

테스트

파일로 남겨서 테스트.

- c++ / c++
- c++ / c#
- c# / c#

개선

- 변경되지 않은 파일들은 json에서 AST 로딩 하기.
- 생성된 파일 컨벤션 옵션 주기