htScheme

A structured and plugin-based scheme interpreter implementation.

How to use

Prerequisites

- A modern C++ compiler supporting c++11 feature. (gcc4.9.2, gcc5.1, clang3.6 have been tested on Linux)
- GNU Make (Make v4.0 has been tested on Linux)

Make

Enter the `scheme` directory and run the following commands to generate various targets:

Command	Function
`make`=`make all`	Call everything below except `dep` and `clean`
`make cli`	Generate `cli` (the command-line interpreter frontend)
`make dep`	Generate `dep.d` which contains the dependencies of files
`make clean`	Remove all files generated by `make`
`make preprocessortest`	Generate `preprocessortest`
`make tokenizertest`	Generate `tokenizertest`
`make asttest`	Generate `asttest`
`make parserstest`	Generate `parserstest`
`make biginttest`	Generate `biginttest`
`make rationaltypetest`	Generate `rationaltypetest`

You may notice that the compilation is rather slow, therefore you can add `-j4` to `make` command in order to parallel the compilation with four threads.

How to develop with htScheme

htScheme has been designed as an extensible architecture of scheme-like languages, thus new types of tokens as well as parsers could be easily added into this program.

Brief introduction to files

preprocessor.hpp/cpp

```
class SchemeUnit
{
    public:
        SchemeUnit(std::istream& schemeStream);
        std::vector<std::string> lines;
        void preprocess(std::istream& schemeStream);
};
```

Accept a `std::istream` as the parameter, then read lines from it until `schemeStream.eof()` and remove the comments with the result stored in `lines`.

tokenizer.hpp/cpp

```
class Tokenizer
{
    public:
        Tokenizer(const std::vector<std::string>& lines);
        void split(const std::vector<std::string>& lines);
        void parse(const std::list<std::string>& rawTokens);
        std::list<std::string> rawTokens;
        std::list<Token> tokens;
        bool complete;
};
```

Accept lines of program, Then `Tokenizer::Tokenizer` will call `Tokenizer:split` and `Tokernizer::parse` in order.

`Tokenizer::split` splits `lines` into several small string pieces stored in `Tokenizer::rawTokens`. For example, `(string-ith "123 34" 2)` will be split into

```
(
string-ith
"123 34"
2
)
```

`Tokenizer::parse` convert `Tokenizer::rawTokens` to `Tokenizer::tokens`.

`Token` is defined as followed in `types/all.hpp`:

```
struct Token
{
    TokenType tokenType; //enum TokenType {OpPlus, ...}
    InfoTypes info; //typedef boost::variant<InfoType1, ...> InfoTypes
    std::string raw; //raw token
};
```

There is an extra variable `Tokenizer::complete` in this class, which represents whether there is no incomplete brackets or quotaion marks. This could be useful in building command-line interpreter. `Tokenizer::complete` can be set by both `Tokenizer::split` and `Tokenizer::parse`.

ast.hpp/cpp

```
class AST
{
    public:
        PASTNode astHead; //typedef std::shared_ptr<ASTNode> PASTNode
        void buildAST(const std::list<Token> &tokens);
        AST (const std::list<Token> &tokens);
        AST();
        friend std::ostream& operator << (std::ostream& o, const AST& ast);
};
```

Build an AST which could be accessed through `AST.astHead` with `std::list<Token>`.

Here is the definition of `ASTNode`:

```
struct ASTNode
{
    NodeType type; //enum NodeType {Bracket, Simple};
    Token token;
    PASTNode parent;
    std::list<PASTNode> ch;

    ASTNode* add(const ASTNode& node); //ch.push_back(std::make_shared<ASTNode>(ASTNode(nod
    void remove(); //Recursively remove all its children then clear ch
};
```

For example, `(+ 2.7 (- 5.6 2.1) 3)` will be converted to the following AST:

```
Type:0 Token.info:0 TokenType:0 //astHead
+----Type:Bracket Token.info:0 TokenType:0
          +---Type:Simple Token.info:0 TokenType:OpPlus
          |---Type:Simple Token.info:2.7 TokenType:Float
          |---Type:Bracket Token.info:0 TokenType:0
```

```
|      +---Type:Simple Token.info:0 TokenType:OpMinus
|      |---Type:Simple Token.info:5.6 TokenType:Float
|      |---Type:Simple Token.info:2.1 TokenType:Float
|---Type:Simple Token.info:3 TokenType:Float
```

parsers.hpp The main part of `parsers.hpp` is in `parsers/all.hpp`

```
class ParsersHelper
{
    ParsersHelper();
    void parse(PASTNode astnode);
};
```

Provide a smart pointer of `ASTNode` to an instance of `ParsersHelper::parse`, then `ParsersHelper` will call according `xxxASTParser::parse(PASTNode parent, ParsersHelper& helper)` to recursively calculate the result of a subtree of AST with its root as `astnode`. After `ParsersHelper::parse`, `astnode.type` will become `Simple`. If `astnode` is already a `Simple` node, nothing will be done.

The parsed version of the above AST is (by calling `parse( **ast.headNode.ch.begin() )`:

```
Type:1 Token.info:0 TokenType:0 //headNode
+----Type:Simple Token.info:9.2 TokenType:Float
```

Warning –DO NOT– directly call `helper.parse(*ch[xx])` in `xxxASTParser::parse(ASTNode& parent, ParsersHelper& helper)` ! Instead, you should copy construct a new `ParsersHelper` to parse its children.

Add your own Token Parser

> An token parser is a struct with static member functions which judge whether a string is a token of this type then convert it to InfoType

In this section, we will try to add a new `Rational` type of token.

Step1: Register the Rational Type Open `types/arch.hpp`, add `Rational` to `enum TokenType{ ... , Rational }`

Step2: Register the Rational Parser

- Open `types/all.hpp`, add `RationalParser` to `#define PARSERS_TUPLE (..., RationalParser)`
- `#include "rational.hpp"`

Step3: Write the Header

- Create `types/rational.hpp`, then include your declaration of your `RationalType` and `"arch.hpp"`
- Use the macro in `arch.hpp` to generate the declaration of your `RationalParser : PARSER_DECLARATION(RationalParser, Rational, RationalType)`

Step4: Implement the Parser

- Create `types/rational.cpp`, then `#include "rational.hpp"` and implement your `RationalType` in it.
- Implement `bool RationalParser::judge(const std::string& token)` which returns whether a token is a rational token and `RationalParser::InfoType RationalParser::get(const std::string& token)` which converts the token to `RationalParser::InfoType`(aka `RationalType`)
- `const TokenType RationalParser::type = Rational;`

Add your own AST Parser

An AST parser is a struct inheritating `ASTParser`, which judges whether it can parse a subtree of AST then parse it.

Note: In token parser there is only static functions, but an AST parser will be instantiated before we use it, therefore it could include some data members (e.g a database)