Writing a LALR parser

helios (12666) Jun 14, 2009 at 7:53am

Today's *Simple Interpreter* thread inspired me to write one.
Since this is the first time I do something like this (well, actually, I had written an expression parser with lookahead before, but it was rather a mess, so I ended up scrapping it), I would like some feedback on how I'm doing.
Particularly whether my design will make things more difficult later on when I start adding more operators.
I intend to only keep one Token at a time (not counting the stack), in the future, so don't bother pointing out the std::deque.

```
#include <iostream>
#include <sstream>
#include <cctype>
#include <deque>
#include <stack>
#include <vector>

struct Token{
    enum TokenType{
        END,
        INTEGER,
        PLUS,
        MINUS,
    } type;
    long intValue;
    Token(TokenType type=END):type(type),intValue(0){}
    Token(long val):type(INTEGER),intValue(val){}
    Token(char character){
        //...
    }
};

class NonTerminal{
    enum NonTerminalType{
        terminal,
        expr,
    } type;
    NonTerminal *trunk;
    std::vector<NonTerminal> branches;
    bool reduced;
public:
    Token leaf;
private:
    void reduce_terminal(){
        this->reduced=1;
        switch (this->leaf.type){
            case Token::INTEGER:
                this->type=expr;
        }
    }
    void reduce_expr(){
        if (!this->branches.size())
            return;
        if (this->branches.size()<3)
            this->leaf=this->branches[0].leaf;
        else{
            this->leaf.type=Token::INTEGER;
            switch (this->branches[1].leaf.type){
                case Token::PLUS:
                    this->leaf.intValue=this->branches[0].leaf.intValue+this->branches[2].leaf.intValue;
                    break;
                case Token::MINUS:
                    this->leaf.intValue=this->branches[0].leaf.intValue-this->branches[2].leaf.intValue;
                    break;
                default:;
            }
        }
        this->reduced=1;
        this->branches.clear();
    }
public:
    NonTerminal(NonTerminal *trunk=0){
        this->type=expr;
        this->trunk=trunk;
        this->reduced=0;
    }
    NonTerminal(const Token &token,NonTerminal *trunk=0){
        this->leaf=token;
        this->type=terminal;
        this->trunk=trunk;
        this->reduced=0;
    }
    void set(const Token &token){
        this->leaf=token;
        this->type=terminal;
        this->trunk=trunk;
        this->reduced=0;
    }
    void push(const Token &token){
        if (this->type==terminal)
            return;
        this->branches.push_back(NonTerminal(token));
    }
    NonTerminal *newBranch(const Token &token){
        this->branches.push_back(NonTerminal(this));
        return &this->branches.back();
```

```
    }
    bool isComplete(){
        if (this->type==terminal)
            return 1;
        if (!this->branches.size())
            return 0;
        for (unsigned a=0;a<this->branches.size();a++)
            if (this->branches[a].type!=terminal && !this->branches[a].reduced)
                return 0;
        switch (this->branches.size()){
            case 1:
                return this->branches[0].leaf.type==Token::INTEGER;
            case 3:
                if (this->branches[0].leaf.type!=Token::INTEGER ||
                        this->branches[1].leaf.type!=Token::PLUS &&
                        this->branches[1].leaf.type!=Token::MINUS ||
                        this->branches[2].leaf.type!=Token::INTEGER)
                    return 0;
                return 1;
            default:
                return 0;
        }
    }
    NonTerminal *reduce(){
        if (!this->isComplete())
            return 0;
        switch (this->type){
            case terminal:
                this->reduce_terminal();
                break;
            case expr:
                this->reduce_expr();
                break;
        }
        return this->trunk?this->trunk:this;
    }
};

class Parser{
    std::deque<Token> expression;
    std::stack<Token> stack;
    long result;
    unsigned state;
    NonTerminal tree,
        *current_node;
    Token read(std::stringstream &stream){
        //(lexer)
    }
    unsigned run_state(){
        Token::TokenType front_token_type;
        switch (this->state){
            case 0:
                front_token_type=this->expression.front().type;
                if (front_token_type==Token::INTEGER)
                    return 3;
                if (front_token_type==Token::END)
                    return 1;
                return 2;
            case 3:
                this->current_node->push(this->expression.front());
                this->expression.pop_front();
                if (this->current_node->isComplete())
                    this->current_node=this->current_node->reduce();
                front_token_type=this->expression.front().type;
                if (front_token_type==Token::PLUS ||
                        front_token_type==Token::MINUS)
                    return 4;
                if (front_token_type==Token::END)
                    return 1;
                return 2;
            case 4:
                this->current_node->push(this->expression.front());
                this->expression.pop_front();
                front_token_type=this->expression.front().type;
                if (front_token_type==Token::INTEGER)
                    return 3;
                return 2;
            default:
                return this->state;
        }
    }
    //1: continue, 0: accept, -1: abort
    int to_state(unsigned n){
        this->state=n;
        switch (n){
            case 1:
                return 0;
            case 2:
                return -1;
            default:
                return 1;
        }
    }
public:
    Parser(const std::string &str){
        std::stringstream stream(str);
        do
            this->expression.push_back(this->read(stream));
        while (this->expression.back().type!=Token::END);
        this->result=0;
        this->state=0;
        this->current_node=&this->tree;
    }
    bool eval(long &res){
```

```
        int ret;
        while ((ret=this->to_state(this->run_state()))==1);
        if (!ret){
            this->tree.reduce();
            res=this->tree.leaf.intValue;
        }
        return !ret;
    }
};

int main(){
    Parser evaluator("12+3-2");
    long res;
    std::cout <<evaluator.eval(res)<<std::endl;
    std::cout <<res<<std::endl;
    return 0;
}
```

Last edited on Jun 14, 2009 at 7:56am

tition (870) Jun 14, 2009 at 6:44pm

What was your motivation of chosing this type of evaluation instead of the obvious recursion?

What I mean is, why didn't you do something along the lines

```
1 void NonTerminal::reduce_expr()
2 { if (!this->branches.size())
3     return;
4   if (this->branches.size()<3)
5     this->leaf=this->branches[0].leaf;//<-I don't get this one...
6     //is it possible that you are forbidding expressions such as "-1"?
7     //if you mean here that if you have less than two leaves, then you
8     //must have one leaf, then I would suggest putting one reassuring
9     //assert(this->branches.size()==1);
10  else
11  { this->leaf.type=Token::INTEGER;
12    switch (this->branches[1].leaf.type)
13    { case Token::PLUS://my main comment is here. Why not recursion?
14                                  //suggestion follows
15        if (!this->branches[0].reduced)
16          this->branches[0].reduce_expr();
17        if (!this->branches[2].reduced)
18          this->branches[2].reduce_expr();
19        //end of suggestion
20        this->leaf.intValue=this->branches[0].leaf.intValue+this->branches[2].leaf.intValue;
21      break;
22      case Token::MINUS://you can use recursion here too
23        this->leaf.intValue=this->branches[0].leaf.intValue-this->branches[2].leaf.intValue;
24      break;
25     default:;
26    }
27  }
28  this->reduced=1;
29  this->branches.clear();
30 }
```

Also, why do you use `int` for reduced instead of `bool`? Do you plan on assigning more than two values to reduced?

Last edited on Jun 14, 2009 at 6:50pm

helios (12666) Jun 14, 2009 at 7:11pm

I'm interested in the state machine aspect. Also, my last parser was recursive, so I thought I'd try something different, this time. Plus, a recursive parser is limited by the size of the stack, while an iterative isn't.

Line 5: Sign inversion is a completely different operand from subtraction. Aside from the obvious, sign inversion is right-associative, while subtraction is left-associative. So in short, yes, I am forbidding it.
Lines 13-19: reduce_expr() assumes that the branches in the current node are already reduced.

And I don't know where you're looking, but NonTerminal::reduced is a bool.

After my previous post, I read a bit of Bison's manual and realized I need either the stack or the tree. Since the stack is simpler to implement, that's what I'll use. Well, actually, I'll use a vector as a stack, since I'll have to take a look at elements other than the top one.

Last edited on Jun 14, 2009 at 7:18pm

tition (870) Jun 14, 2009 at 9:22pm

Hehe you inspired me to try write my own parser as well. I might also need it one day too!

Will post it as soon as I am done (it is now 60% ready).

Cheers!

helios (12666) Jun 15, 2009 at 12:36am

There we go.

```
#include <cctype>
#include <cstdarg>
#include <cmath>
#include <iostream>
#include <sstream>
#include <stack>
#include <vector>

struct Token{
        enum TokenType{
                null=0,
```

```
        END=1,
        INTEGER='0',
        PLUS='+',
        MINUS='-',
        MUL='*',
        DIV='/',
        POW='^',
        LPAREN='(',
        RPAREN=')',
        expr=128
    } type;
    double intValue;
    Token(TokenType type=END):type(type),intValue(0){}
    Token(long val):type(INTEGER),intValue(val){}
    Token(char character){
        this->type=(TokenType)character;
    }
};

struct Rule{
    Token reduces_to;
    std::vector<Token> constraints;
    Token lookahead;
    Rule(const Token &to,const Token &la,unsigned constraints,...){
        this->reduces_to=to;
        this->lookahead=la;
        va_list list;
        va_start(list,constraints);
        this->constraints.reserve(constraints);
        for (unsigned a=0;a<constraints;a++)
            this->constraints.push_back(va_arg(list,Token::TokenType));
    }
    bool matches(const std::vector<Token> &stack,const Token &lookahead){
        if (stack.size()<this->constraints.size() ||
                this->lookahead.type!=Token::null && this->lookahead.type!=lookahead.type)
            return 0;
        const Token *array=&stack[stack.size()-this->constraints.size()];
        for (unsigned a=0,size=this->constraints.size();a<size;a++)
            if (array[a].type!=this->constraints[a].type)
                return 0;
        return 1;
    }
};

class Parser{
    std::stringstream stream;
    std::vector<Token> stack;
    bool result;
    std::vector<Rule> rules;
    Token read(){
        char character;
        while (!this->stream.eof() && isspace(character=this->stream.peek()))
            this->stream.get();
        if (this->stream.eof())
            return Token::END;
        character=this->stream.peek();
        if (isdigit(character)){
            std::string str;
            str.push_back(this->stream.get());
            while (isdigit(this->stream.peek()))
                str.push_back(this->stream.get());
            long temp=atol(str.c_str());
            return temp;
        }
        return (char)this->stream.get();
    }
    bool reduce(const Token &lookahead){
        long rule_index=-1;
        unsigned max=0;
        for (unsigned a=0;a<this->rules.size();a++){
            if (this->rules[a].matches(this->stack,lookahead) && this->rules[a].constraints.size()>max){
                rule_index=a;
                max=this->rules[a].constraints.size();
            }
        }
        if (rule_index<0 || this->rules[rule_index].reduces_to.type==Token::null)
            return 0;
        Rule &rule=this->rules[rule_index];
        Token new_token(rule.reduces_to);
        Token *redex=&this->stack[this->stack.size()-rule.constraints.size()];
        switch (rule_index){
            case 0: //expr <- INTEGER
                new_token.intValue=redex[0].intValue;
                break;
            case 1: //expr <- '(' expr ')'
            case 2: //expr <- '+' expr
                new_token.intValue=redex[1].intValue;
                break;
            case 3: //expr <- '-' expr
                new_token.intValue=-redex[1].intValue;
                break;
            case 4: //impossible
            case 5: //expr <- expr '^' expr
                new_token.intValue=pow((double)redex[0].intValue,(double)redex[2].intValue);
                break;
            case 6: //expr <- expr '*' expr
                new_token.intValue=redex[0].intValue*redex[2].intValue;
                break;
            case 7: //expr <- expr '/' expr
                new_token.intValue=redex[0].intValue/redex[2].intValue;
                break;
            case 8: //impossible
            case 9: //impossible
            case 10: //expr <- expr '+' expr
```

```
                                new_token.intValue=redex[0].intValue+redex[2].intValue;
                                break;
                        case 11: //impossible
                        case 12: //impossible
                        case 13: //expr <- expr '-' expr
                                new_token.intValue=redex[0].intValue-redex[2].intValue;
                                break;
                }
                for (unsigned a=0;a<rule.constraints.size();a++)
                        this->stack.pop_back();
                this->stack.push_back(new_token);
                return 1;
        }
        bool run_state(){
                Token next_token=this->read();
                while (this->reduce(next_token));
                switch (next_token.type){
                        case Token::END:
                                this->result=(this->stack.size()==1);
                                return 0;
                        case Token::INTEGER:
                        case Token::PLUS:
                        case Token::MINUS:
                        case Token::MUL:
                        case Token::DIV:
                        case Token::RPAREN:
                        case Token::LPAREN:
                        case Token::POW:
                                this->stack.push_back(next_token);
                                return 1;
                        default:
                                this->result=0;
                                return 0;
                }
        }
        void initializeRules(){
                this->rules.clear();
                /*rule 0*/      this->rules.push_back(Rule(     Token::expr,    Token::null,    1,      Token::INTEGER  ));

                /*rule 1*/      this->rules.push_back(Rule(     Token::expr,    Token::null,    3,      Token::LPAREN,  Toke

                /*rule 2*/      this->rules.push_back(Rule(     Token::expr,    Token::null,    2,      Token::PLUS,    Toke
                /*rule 3*/      this->rules.push_back(Rule(     Token::expr,    Token::null,    2,      Token::MINUS,   Toke

                /*rule 4*/      this->rules.push_back(Rule(     Token::null,    Token::POW,             3,      Token::expr,
                /*rule 5*/      this->rules.push_back(Rule(     Token::expr,    Token::null,    3,      Token::expr,    Toke

                /*rule 6*/      this->rules.push_back(Rule(     Token::expr,    Token::null,    3,      Token::expr,    Toke

                /*rule 7*/      this->rules.push_back(Rule(     Token::expr,    Token::null,    3,      Token::expr,    Toke

                /*rule 8*/      this->rules.push_back(Rule(     Token::null,    Token::MUL,             3,      Token::expr,
                /*rule 9*/      this->rules.push_back(Rule(     Token::null,    Token::DIV,             3,      Token::expr,
                /*rule 10*/     this->rules.push_back(Rule(     Token::expr,    Token::null,    3,      Token::expr,    Toke

                /*rule 11*/     this->rules.push_back(Rule(     Token::null,    Token::MUL,             3,      Token::expr,
                /*rule 12*/     this->rules.push_back(Rule(     Token::null,    Token::DIV,             3,      Token::expr,
                /*rule 13*/     this->rules.push_back(Rule(     Token::expr,    Token::null,    3,      Token::expr,    Toke
        }
public:
        Parser(const std::string &str)
                        :stream(str){
                this->result=0;
                this->initializeRules();
        }
        bool eval(double &res){
                while (this->run_state());
                if (this->result)
                        res=this->stack.front().intValue;
                else
                        this->stack.clear();
                return this->result;
        }
};

int main(){
        Parser evaluator("2^2^2");
        double res=0;
        std::cout <<(evaluator.eval(res)?"ACCEPT":"ABORT")<<std::endl;
        std::cout <<res<<std::endl;
        return 0;
}
```

I can implement both precedence and associativity. Although I don't know how lack of associativity works.
It doesn't run on a state machine. Instead of adding more states, I add more rules. For example, an exponentiation can only be reduced if the next token is not ^.
It's still technically a LALR, though, since is looks ahead and runs for left to right (or at least I *think* it is).

EDIT: By the way, initializeRules() looks good with a 4 columns tab.

Last edited on Jun 15, 2009 at 1:10am

tition (870) Jun 15, 2009 at 2:59am

Here is my version :)

Let us exchange ideas :) I will be a bit delayed with replies though cause Real Life Work is calling me :((((

[Edit: Fixed mistakes. This is 3rd version already]
I haven't really tested most of it (except for the expression i put in there), but unless I messed it up it should support brackets, +,-,/,*. It chops the expression non-recursively, but computes recursively cause I was too lazy to introduce

a total order to the generated tree :(

```
#include <iostream>
#include <vector>
#include <assert.h>

class Expression;

class Tokens
{
public:
        static const char PLUS='+';
        static const char MINUS='-';
        static const char MUL='*';
        static const char DIV='/';
        static const char OpenBracket= '(';
        static const char ClosingBracket=')';
        static bool IsAnOperationToken(char x);
        static bool IsABracketToken(char x);
        static bool IsAnUnaryOperationToken(char x);
        static bool IsADataToken(char x);
        static int AssociativityWeakness(char x, bool isUnary);
};

bool Tokens::IsABracketToken(char x)
{ return x==Tokens::OpenBracket ||
                                 x==Tokens::ClosingBracket;
}

bool Tokens::IsAnOperationToken(char x)
{ return x==Tokens::PLUS ||
                                 x==Tokens::MINUS ||
                                 x==Tokens::MUL ||
                                 x==Tokens::DIV;
}

bool Tokens::IsAnUnaryOperationToken(char x)
{ return x==Tokens::PLUS ||
                                 x==Tokens::MINUS;
}

bool Tokens::IsADataToken(char x)
{ return !( Tokens::IsAnOperationToken(x)
                                          ||
                                                Tokens::IsABracketToken(x)
                                        );
}

int Tokens::AssociativityWeakness(char x, bool isUnary)
{ assert(Tokens::IsAnOperationToken(x));
        if (!isUnary)
        {       if (x==Tokens::PLUS ||
                                x==Tokens::MINUS)
                        return 10;
                if (x==Tokens::MUL ||
                                x==Tokens::DIV)
                        return 5;
        }
        else
        { if (x==Tokens::PLUS ||
                                x==Tokens::MINUS)
                                return 7;
        }
        return -1;
}

class ExpressionLeaf
{
private:
        friend class Expression;
        Expression* BossExpression;
        int leftIndex;
        int rightIndex;
        ExpressionLeaf* leftLeaf;
        ExpressionLeaf* rightLeaf;
        char OperationBetweenLeftAndRightLeaf;
        bool ComputeSuccessorLeaves();//returns true if the expression is reduced
        bool SplitInTwo(int operationIndex);
        int ComputeRecursively();
        ExpressionLeaf()
        {       this->leftLeaf=0;
                this->rightLeaf=0;
                this->OperationBetweenLeftAndRightLeaf=0;
        }
};

class Expression: public std::vector<ExpressionLeaf*>
{
public:
        int FindIndexClosingBracket(int IndexOpeningBracket);
        void Chop();
        void init();
        int Compute()
        { this->init();
                this->Chop();
                if (this->theStringToBeChopped.size()!=0)
                        return this->operator [](0)->ComputeRecursively();
                else
                        return 0;
        }
        ~Expression()
        { for (int i=0;i<this->size();i++)
```

```
                { delete this->operator[](i);
                }
        };
        std::string theStringToBeChopped;
};

int Expression::FindIndexClosingBracket(int IndexOpeningBracket)
{ assert(this->theStringToBeChopped[IndexOpeningBracket]==Tokens::OpenBracket);
        unsigned int NumOpeningBrackets=1;
        unsigned int NumClosingBrackets=0;
        unsigned int i=IndexOpeningBracket;
        while(NumOpeningBrackets>NumClosingBrackets && i<this->theStringToBeChopped.size())
        {       i++;
                if (this->theStringToBeChopped[i]==Tokens::OpenBracket)
                        NumOpeningBrackets++;
                if (this->theStringToBeChopped[i]==Tokens::ClosingBracket)
                        NumClosingBrackets++;
        }
        assert(i<this->theStringToBeChopped.size());
        return i;
}

void Expression::init()
{ this->resize(1);
        this->operator [](0)= new ExpressionLeaf;
        this->operator [](0)->leftIndex=0;
        this->operator [](0)->rightIndex=this->theStringToBeChopped.size()-1;
        this->operator [](0)->BossExpression=this;
}

void Expression::Chop()
{ int currentIndex= 0;
        while (currentIndex<this->size())
        { if (this->operator [](currentIndex)->ComputeSuccessorLeaves())
                        currentIndex++;
        }
}

int ExpressionLeaf::ComputeRecursively()
{ if(this->leftLeaf==0 && this->rightLeaf==0)
        { std::string tempS;
                tempS= this->BossExpression->theStringToBeChopped.substr(this->leftIndex, this->rightIndex - this->leftIndex
                return std::atoi(tempS.c_str());
        }
        else
        { if (this->leftLeaf==0)
                { if (this->OperationBetweenLeftAndRightLeaf==Tokens::MINUS)
                                return - this->rightLeaf->ComputeRecursively();
                        else
                                return this->rightLeaf->ComputeRecursively();
                }
                else
                { switch(this->OperationBetweenLeftAndRightLeaf)
                        {
                        case Tokens::MINUS:
                                return this->leftLeaf->ComputeRecursively()- this->rightLeaf->ComputeRecursively();
                        break;
                        case Tokens::PLUS:
                                return this->leftLeaf->ComputeRecursively() + this->rightLeaf->ComputeRecursively();
                        case Tokens::MUL:
                                return this->leftLeaf->ComputeRecursively()*this->rightLeaf->ComputeRecursively();
                        case Tokens::DIV:
                                return this->leftLeaf->ComputeRecursively()/ this->rightLeaf->ComputeRecursively();
                        default:
                                return 0;
                        }
                }
        }
}

//the return type is to facilitate an error catching mechanism
//different from my favourite assert
bool ExpressionLeaf::SplitInTwo(int operationIndex)
{ assert(operationIndex!=this->rightIndex);
        this->OperationBetweenLeftAndRightLeaf=this->BossExpression->theStringToBeChopped[operationIndex];
        if (operationIndex== this->leftIndex)
        { //unary operations are allowed. We simply set the left leaf to be zero/
                //we need to check for unary operation abuse however. Turn off if you want to allow it
                //(for example if you think --1 is allowed and is the same as -(-1))
                if (this->leftIndex>0)
                { assert(!Tokens::IsAnOperationToken(this->BossExpression->theStringToBeChopped[this->leftIndex-1]));
                }
                this->leftLeaf=0;
                this->rightLeaf= new ExpressionLeaf;
                this->rightLeaf->BossExpression = this->BossExpression;
                this->rightLeaf->leftIndex= this->leftIndex+1;
                this->rightLeaf->rightIndex= this->rightIndex;
                this->BossExpression->push_back(this->rightLeaf);
                return true;
        }
        this->leftLeaf  = new ExpressionLeaf;
        this->rightLeaf = new ExpressionLeaf;
        this->leftLeaf->leftIndex= this->leftIndex;
        this->leftLeaf->rightIndex= operationIndex-1;
        this->rightLeaf->rightIndex= this->rightIndex;
        this->rightLeaf->leftIndex= operationIndex+1;
        this->leftLeaf->BossExpression= this->BossExpression;
        this->rightLeaf->BossExpression= this->BossExpression;
        this->BossExpression->push_back(this->leftLeaf);
        this->BossExpression->push_back(this->rightLeaf);
        return true;
}

//returns true if the expression gets split or is reduced
```

```
//false otherwise
bool ExpressionLeaf::ComputeSuccessorLeaves()
{ if (this->leftIndex>this->rightIndex)
        { return true;
        }
        if (this->BossExpression->theStringToBeChopped[this->leftIndex]==Tokens::OpenBracket)
        {       int closingBracketIndex=this->BossExpression->FindIndexClosingBracket(this->leftIndex);
                //we gotta check whether our expression is of the type (1+2)
                if (closingBracketIndex==this->rightIndex)
                { this->leftIndex++;
                        this->rightIndex--;
                        return false;
                }
                //our expression is of the type (1+2)+3
                this->SplitInTwo(closingBracketIndex+1);
                return true;
        }
//label: find operation not enclosed by brackets
        int theOperationIndex=-1;
        int currentAssociativityWeakness=-1;
        int NumOpenBrackets=0;
        int NumClosedBrackets=0;
        for (int i=this->leftIndex;i<this->rightIndex;i++)
        {       char operationCandidate=this->BossExpression->theStringToBeChopped[i];
                if (operationCandidate==Tokens::OpenBracket)
                        NumOpenBrackets++;
                if (operationCandidate==Tokens::ClosingBracket)
                        NumClosedBrackets++;
                if (Tokens::IsAnOperationToken(operationCandidate)&& NumOpenBrackets==NumClosedBrackets)
                { int candidateAssociativityWeakness=
                                                Tokens::AssociativityWeakness
                                                        (operationCandidate,i==this->leftIndex);
                        if (candidateAssociativityWeakness>currentAssociativityWeakness)
                        { theOperationIndex=i;
                                currentAssociativityWeakness=candidateAssociativityWeakness;
                        }
                }
        }
//label: end of search
        if (theOperationIndex==-1)
                return true;
        this->SplitInTwo(theOperationIndex);
        return true;
}

void main()
{ Expression x;
        x.theStringToBeChopped= "-(12+2*(-8*6+5*4)+13+19)*2";
        std::cout <<x.Compute();
        int a;
        std::cin>>a;
        return;
}
```

Last edited on Jun 15, 2009 at 4:38am

helios (12666) Jun 15, 2009 at 3:14am

Wait, you actually use assert() for error handling? I suppose you don't know that compiling for release disables all assert()s.

EDIT: Oh, by the way. The containers in the standard library are not designed to be used as base classes.

Last edited on Jun 15, 2009 at 3:21am

tition (870) Jun 15, 2009 at 3:28am

Fixed the mistakes I know.

You supposed wrong, I know release disables assert. That is why I left functionality out for real error handling: `bool ExpressionLeaf::SplitInTwo(int operationIndex)` for the time being returns only true. Since it is the memory allocation unit it is where errors should be raised, by returning false. However, I did not program anything to handle errors yet, so it better be left with assert and used with Debug compiling.

I didn't know that for the standard library...

Last edited on Jun 15, 2009 at 5:30am

tition (870) Jun 15, 2009 at 4:50am

So can you explain more on your concept?

As far as mine goes, it is the following:

0. I chose the generate-tree approach. However, I store the `ExpressionLeaf*` of my tree in a vector, so it is a "hybrid" approach.
1. I keep the original expression's string in memory. All other references to it are made by providing starting index (`int leftIndex`) and ending index (`int rightIndex`).
2. I realize a simple routine which computes for a given open bracket its counterpart closing bracket `int Expression::FindIndexClosingBracket(int IndexOpeningBracket)`
3. I generate the tree by setting simple rules for splitting an expression.

Parsing
3.0 Start.
3.1 If an expression starts with an open bracket whose closing bracket is the expression's end, I "remove" the brackets(shift leftIndex and rightIndex) and go back one step; else I proceed.
3.2 If an expression doesn't fall in the category described in 3.1, it is obvious that either 1) it is an atomic expression (can't do anything with it - say, a constant) or 2) there must be an operation token some place that is not enclosed by brackets. The cycle after `//label: find operation not enclosed by brackets` finds that operation token if it exists.
3.2.1 Important note. When finding intermediate operation tokens, one must be careful for the order of precedence of

operators. For example, in a*b+c, a valid split of the expression is add(mult(a,b),c), which means that the in step 3.2 we are allowed to pick only the '+' token. That is what all the `int Tokens::AssociativityWeakness(char x, bool isUnary)` jazz is all about.
3.3 a) If an intermediate operation token is found, we split the expression with `bool ExpressionLeaf::SplitInTwo(int operationIndex)`. The newly created expressions (`ExpressionLeaf`) are recorded in our global `Expression`.
3.3 b) If no intermediate operation token is found ("atomic expression case") we "mark" the expression as reduced by returning with a true.

4. We execute step 3 to all non-atomic expressions. Note that the function return values are set so that one doesn't actually have to keep a `bool isReduced` member of `ExpressionLeaf`.

***
So that was the parser. Once you have the tree structure, evaluating it recursively is a piece of cake. (`int ExpressionLeaf::ComputeRecursively()`)

Last edited on Jun 15, 2009 at 5:27am

helios (12666) Jun 15, 2009 at 6:38am

Well, it's very simple, really.
The main components are the rule list and the reduce() function.

Each rule in the rule list specifies what will the top of the stack be reduced to if it matches a list of constraints and the lookahead token matches a type. The rule may also specify that the lookahead token can be of any type and that the stack should not be reduced if it matches that rule.

For example, one of the rules (rule 10) says that if the top of the stack contains an expression, a +, and another expression and the lookahead token is a *, then the stack should not be reduced. On the other hand, another rule (rule 6) says that a stack containing expr '*' expr should be reduced to an expr regardless of what the lookahead token is.

The reduce function finds which rule to use to reduce the stack by looking for [the biggest rule that matches the top of the stack and the lookahead token]. If there are two rules (i.e. both rules have constraints of the same size) that meet this condition, it will choose the first one it finds, so the order of the rules is crucial.
If it doesn't find a match or the match specifies that the stack should not be reduced, reduce quits. Otherwise, the rule is executed, the stack is popped and then pushed back with the new non-terminal.

run_state() (actually, I should rename the function) is pretty self-explanatory.

Example:
2^2^2+1+2*3
INTEGER POW INTEGER POW INTEGER PLUS INTEGER PLUS INTEGER MUL INTEGER
Stack: <empty>
Can't reduce further
Shift INTEGER

Stack: INTEGER
Reduce with (expr -> INTEGER): {expr|INTEGER}
Can't reduce further
Shift POW

Stack: expr POW
Can't reduce further
Shift INTEGER

Stack: expr POW INTEGER
Reduce with (expr -> INTEGER): expr POW {expr|INTEGER}
Can't reduce with (expr -> expr POW expr) because lookahead is POW
Can't reduce further
shift POW

Stack: expr POW expr POW
Can't reduce further
Shift INTEGER

Stack: expr POW expr POW INTEGER
Reduce with (expr -> INTEGER): expr POW expr POW {expr|INTEGER}
Can reduce with (expr -> expr POW expr) because lookahead is not POW: expr POW {expr|expr POW expr}
Can reduce with (expr -> expr POW expr) because lookahead is not POW: {expr|expr POW expr}
Can't reduce further
shift PLUS

Stack: expr PLUS
Can't reduce further
Shift INTEGER

Stack: expr PLUS INTEGER
Reduce with (expr -> INTEGER): expr PLUS {expr|INTEGER}
Can't reduce with (expr -> expr PLUS expr) because lookahead is MUL
Can't reduce further
shift MUL

Stack: expr PLUS expr MUL
Can't reduce further
shift INTEGER

Stack: expr PLUS expr MUL INTEGER
Reduce with (expr -> INTEGER): expr PLUS expr MUL {expr|INTEGER}
Reduce with (expr -> expr MUL expr): expr PLUS {expr|expr MUL expr}
Can reduce with (expr -> expr PLUS expr) because lookahead is not MUL: {expr|expr PLUS expr}
Can't reduce further
Look ahead is END
The stack length is exactly 1, so were no errors.

My approach is simplistic, which is a plus, but unlike a state machine, it can only detect that *some* error has occurred, not *where* it occurred, because the error detection is a single at the end of execution.
However, it's good enough to generate at least a simple parser from Yacc-esque rules. Right now I'm working on how

to do that. One of the problems I need to solve is "how do I know (expr '+' expr) and (expr '*' expr) are in conflict when there is no extra precedence information?"

Last edited on Jun 15, 2009 at 8:06am

---

tition (870) Jun 15, 2009 at 8:13am

Can you explain the format of your Rules (with words if possible because I really lost in the syntax:()?
There is a bug with the POW token, 1+2^2=9.

Last edited on Jun 15, 2009 at 8:15am

---

closed account (*S6k9GNh0*) Jun 15, 2009 at 8:30am

Wow. I tried learning the concept of LALR parsers and my brain hurts now. I'll save this for another day lol. It seems to be one of those things that take a bit to digest.

---

helios (12666) Jun 15, 2009 at 8:30am

The first parameter to Rule::Rule() is what the rule reduces to. If Token::null is passed, the parser will not try to reduce. This is used to enforce precedence and associativity.
The second parameter is the constraint on the lookahead token. If Token::null is passed, there's no constraint.
The third parameter is the number of variadic parameters to follow.
From then on are the constraints that will be applied to the stack.

And yeah, you're right. I forgot to add some more rules to PLUS, MINUS, DIV, and MUL. a<anything other than ^>b^c==(a*b)^c, because I accidentally gave ^ the highest precedence. Luckily, fixing this is just a matter of copy-pasting and slightly editing a few lines.

Last edited on Jun 15, 2009 at 8:35am

---

tition (870) Jun 15, 2009 at 9:02am

aha... I think I finally got it:
`/*rule 1*/Rule(Token::expr, Token::null,3,Token::LPAREN, Token::expr, Token::RPAREN )`
means:

if the last three tokens are [LPAREN, expr, RPAREN] (in this order) and if the lookahead is null = arbitrary or there is no lookahead token, then substitute [LPAREN, expr, RPAREN] with expr.

Schematically:

LPAREN, expr, RPAREN --> expr
4th param 5th param 6th param 1st param

2nd param specifies what the lookahead token must be.
3rd param is a technicality.

Great idea! And it is very fast too!
Cheers!

Last edited on Jun 15, 2009 at 9:05am

---

mcleano (922) Jun 16, 2009 at 2:37am

tition, both your codes are way over my head! but your main is of type void, shouldn't it be int?

---

jbrooksuk (30) Jun 16, 2009 at 4:00am

Wow that's a fair bit of code there.

It's a bit complex for my liking but I have a lot to learn from it.

---

closed account (*S6k9GNh0*) Jun 16, 2009 at 4:03am

It's not the code itself that troubles me. It's simply the concept of the LR or LALR parser. Wikipedia gives a mediocre example on how it works. Here's a decent tutorial that puts it in much better terms:

http://www.devincook.com/goldparser/doc/about/lalr.htm

This tutorial doesn't explain non-terminal symbols so here:
http://en.wikipedia.org/wiki/Terminal_symbol

Last edited on Jun 16, 2009 at 4:07am

---

helios (12666) Jun 16, 2009 at 4:20am

It sounds more complicated than it actually is. A good introduction to the subject is generating a parser with Yacc or Bison (that's how I learnt it no more than a month ago).
Bison's manual also helps a great deal to understand how the parser works.

---

tition (870) Jun 16, 2009 at 5:03am

A suggestion to helios. It would be nice to be able to suggest to the user possible mistakes. So, it completely makes sense to build the tree structure underlying the parsed expression. (I have no clue how you would give error suggestions otherwise).

That will be very easy to implement on top of your code: in the `reduce` function, besides evaluating the expressions, you can also build the underlying expression tree structure (from the bottom up).

```
switch (rule_index){
/*....*/
        case 6: //expr <- expr '*' expr
                new_token.intValue=redex[0].intValue*redex[2].intValue;
                //here add code to make a new node of a tree with left successor
                //the left expression, right successor the right expression, and
                //store operation token * in the new node.
```

```
/*etc.*/
}
```

This way, you will build the expression tree structure in a much nicer fashion than I do. This is in fact the main difference between my slow approach and yours - I build the tree "from the top down", and you build it "from the leaves up". Of course both approaches are correct, but mine is O(n^2) and yours is O(n) (where n is the size of the expression to be parsed), which will be quite a difference if n is 1000 :).

It will be nice for me to try to merge a tree structure in your code. I was thinking first of applying your approach to my code but I think you actually did the tougher part, so it would be quicker to just paste stuff to your code. If I find the time to do so I will post the result here (I will note the code I took from you, but will probably rename it to my tastes :).

What I like with your parser is that you actually have no trouble parsing expressions such as "-1" (which is an "unary" operation (i.e. takes only one argument)) - you just add some extra rules (rules 2 and 3 in your code). In the same way you will have no trouble parsing functions with more than two arguments.

Cheers!

P.S.

> but your main is of type void, shouldn't it be int?

I don't know... *scratches head* Umm, why should it be int?

Last edited on Jun 16, 2009 at 5:26am

helios (12666) Jun 16, 2009 at 5:26am

While I agree that it's not hard to replace the stack with a tree (I originally did that in the opposite direction for the sake of simplicity and efficiency), I still very much doubt you'll manage to get a more advance error checking into my design without throwing away some of the generality, which is the point of using rules. The problem with reduce() is that it's unaware of what the rules do. It's either able to reduce, or unable to reduce, and neither case necessarily mean there's been an error.

I'm currently trying to figure out how to generate a syntax table from a set of reduction rules (e.g. any of the rules that don't reduce to null in my code).
The Wikipedia article on LR parsers has helped somewhat. I would prefer if they used full names for terminals and non-terminals in compiler theory, rather than just single letters, though. It'd make things easier to follow.

EDIT: void main() is non-standard.

Last edited on Jun 16, 2009 at 5:26am

Pages: 1 2

Home page | Privacy policy

Spotted an error? contact us