Cython

Relatório II: Análise sintática e recuperação de erros

Matheus S. Pinheiro Bittencourt e Thales A. Zirbel Hübner

17 de Abril de 2018

Universidade Federal de Santa Catarina

1. Bison

Foi utilizado bison para realizar o analisador sintático. O bison realiza a análise sintática com o método de LALR(1) e permite declarar explicitamente relações de antecedência e associatividade, resolvendo problemas da gramática sem necessidade de alterá-la.

Além disso, o bison avisa quando há conflitos de shift/reduce na linguagem e avisa onde estes se encontram na tabela de parsing, facilitando a correção destas ambiguidades.

2. Parser

A seguir está o arquivo do parser em bison da linguagem. Ele contém a gramática criada a invocação e criação dos nodos da árvore sintática, as definições de precedência

```
%language "c++"
%skeleton "lalr1.cc"

%define parser_class_name { cython_parser }
%define api.token.constructor
%define api.value.type variant
%define parse.error verbose

%locations

%code requires
{
#include <string>
#include <list>
#include <lexical_error.h>
#include <ast.h>
}

%code
{
extern FILE* yyin;
extern yy::cython_parser::symbol_type yylex();
extern void yypop_buffer_state();

std::list<ast::node*> program;
}

/* terminal symbols */
%token <bool> BOOL "boolean"
```

```
%token <std::string> IDENTIFIER "identifier"
%token <double> FLOAT_L "float literal"
%token <int> INT_L "integer literal"
%token <std::string> STRING_L "string literal"

%token NL "new line"
%token EOF_T 0 "end of file"

%token DEF "def"
%token IF "if"
%token ELIF "elif"
%token ELSE "else"
%token FOR "for"
%token WHILE "while"
%token DO "do"
%token BEGIN_T "begin"
%token END_T "end"
%token RETURN "return"

%token INT "int"
%token FLOAT "float"
%token CHAR "char"
%token VOID "void"

%token COLON ":"
%token SEMICOLON ";"
%token COMMA ","
%token ARROW "->"
%token AMPERSEND "&"

%token PLUS "+"
%token MINUS "-"
%token TIMES "*"
%token DIV "/"
%token EXP "**"

%token AND "and"
%token OR "or"
%token NOT "not"

%token LT "<"
%token GT ">"
%token LE "<="
%token GE ">="
%token EQ "=="
```

```
%token NE "!="

%token LPAREN "("
%token RPAREN ")"
%token LBRACKET "["
%token RBRACKET "]"

%token ASSIGN "="

/* non-terminal symbols */
%type <ast::block> inner_block block else
%type <ast::node*> line declaration func_declaration expression
atom_expr
%type <ast::node*> statement if_stmt for_stmt while_stmt return_stmt
%type <ast::node*> assignment func_call
%type <std::list<ast::elif_stmt>> elif
%type <ast::name> name
%type <ast::arg> arg
%type <ast::type> type
%type <std::list<ast::arg>> args_list
%type <std::list<ast::node*>> parameters

/* precedence */
%right ASSIGN
%left OR
%left AND
%left NOT
%left GT LT GE LE EQ NE
%left PLUS MINUS
%left TIMES DIV
%right EXP
%left UMINUS

%%

program
      : program_
      | %empty
      ;

program_
      : program_ declaration nl { program.push_back($2); }
      | program_ func_declaration { program.push_back($2); }
      | declaration nl { program.push_back($1); }
      | func_declaration { program.push_back($1); }
```

```
      ;

declaration
      : IDENTIFIER COLON type { $$ = new ast::declaration($1, $3,
nullptr); }
      | IDENTIFIER COLON type ASSIGN expression {
            $$ = new ast::declaration($1, $3, $5);
      }
      ;

func_declaration
      : DEF IDENTIFIER LPAREN args_list RPAREN ARROW type block nl {
            $$ = new ast::func($2, $4, $7, $8);
      }
      | DEF IDENTIFIER LPAREN RPAREN ARROW type block nl {
            $$ = new ast::func($2, $6, $7);
      }
      ;

block
      : BEGIN_T inner_block END_T { $$ = $2; }
      ;

inner_block
      : inner_block line { $1.add_line($2); $$ = $1; }
      | nl line { $$ = ast::block($2); }
      ;

line
      : declaration nl { $$ = $1; }
      | statement nl { $$ = $1; }
      | expression nl { $$ = $1; }
      ;

statement
      : if_stmt { $$ = $1; }
      | for_stmt { $$ = $1; }
      | while_stmt { $$ = $1; }
      | return_stmt { $$ = $1; }
      ;

expression
      : expression PLUS expression {
            $$ = new ast::binary_operation(ast::plus, $1, $3);
      }
```

```
    | expression MINUS expression {
        $$ = new ast::binary_operation(ast::minus, $1, $3);
    }
    | expression TIMES expression {
        $$ = new ast::binary_operation(ast::times, $1, $3);
    }
    | expression DIV expression {
        $$ = new ast::binary_operation(ast::div, $1, $3);
    }
    | expression EXP expression {
        $$ = new ast::binary_operation(ast::exp, $1, $3);
    }
    | expression AND expression {
        $$ = new ast::binary_operation(ast::_and, $1, $3);
    }
    | expression OR expression {
        $$ = new ast::binary_operation(ast::_or, $1, $3);
    }
    | NOT expression { $$ = new ast::unary_operation(ast::_not, $2); }
    | MINUS expression %prec UMINUS {
        $$ = new ast::unary_operation(ast::uminus, $2);
    }
    | expression GT expression {
        $$ = new ast::binary_operation(ast::gt, $1, $3);
    }
    | expression LT expression {
        $$ = new ast::binary_operation(ast::lt, $1, $3);
    }
    | expression GE expression {
        $$ = new ast::binary_operation(ast::ge, $1, $3);
    }
    | expression LE expression {
        $$ = new ast::binary_operation(ast::le, $1, $3);
    }
    | expression EQ expression {
        $$ = new ast::binary_operation(ast::eq, $1, $3);
    }
    | expression NE expression {
        $$ = new ast::binary_operation(ast::ne, $1, $3);
    }
    | LPAREN expression RPAREN { $$ = $2; }
    | assignment { $$ = $1; }
    | atom_expr { $$ = $1; }
    ;
```

```
atom_expr
      : name { $$ = new ast::name($1); }
      | func_call { $$ = $1; }
      | INT_L { $$ = new ast::int_l($1); }
      | FLOAT_L { $$ = new ast::float_l($1); }
      | STRING_L { $$ = new ast::string_l($1); }
      | BOOL { $$ = new ast::bool_l($1); }
      ;

assignment
      : name ASSIGN expression { $$ = new ast::assignment($1, $3); }
      ;

func_call
      : IDENTIFIER LPAREN parameters RPAREN { $$ = new
ast::func_call($1, $3); }
      | IDENTIFIER LPAREN RPAREN { $$ = new ast::func_call($1); }
      ;

parameters
      : parameters COMMA expression { $1.push_back($3); }
      | expression { $$ = {$1}; }
      ;

if_stmt
      : IF expression DO inner_block END_T {
            $$ = new ast::if_stmt(
                  $2, $4, std::list<ast::elif_stmt>(), ast::block());
      }
      | IF expression DO inner_block elif END_T {
            $$ = new ast::if_stmt($2, $4, $5, ast::block());
      }
      | IF expression DO inner_block else END_T {
            $$ = new ast::if_stmt($2, $4, std::list<ast::elif_stmt>(),
$5);
      }
      | IF expression DO inner_block elif else END_T {
            $$ = new ast::if_stmt($2, $4, $5, $6);
      }
      ;

elif
      : elif ELIF expression inner_block {
            $1.push_back(ast::elif_stmt($3, $4));
            $$ = $1;
```

```
      }
      | ELIF expression inner_block {
            $$ = {ast::elif_stmt($2, $3)};
      }
      ;

else
      : ELSE inner_block { $$ = $2; }
      ;

for_stmt
      : FOR declaration SEMICOLON expression SEMICOLON expression block
{
            $$ = new ast::for_stmt($2, $4, $6, $7);
      }
      | FOR expression SEMICOLON expression SEMICOLON expression block {
            $$ = new ast::for_stmt($2, $4, $6, $7);
      }
      ;

while_stmt
      : WHILE expression block { $$ = new ast::while_stmt($2, $3); }
      ;

return_stmt
      : RETURN expression { $$ = new ast::return_stmt($2); }
      ;

args_list
      : args_list COMMA arg { $1.push_back($3); $$ = $1; }
      | arg { $$ = {$1}; }
      ;

arg
      : IDENTIFIER COLON type { $$ = ast::arg($1, $3, false); }
      | IDENTIFIER AMPERSEND COLON type { $$ = ast::arg($1, $4, true); }
      ;

type
      : type LBRACKET RBRACKET { $1.add_dimension(0); $$ = $1; }
      | type LBRACKET INT_L RBRACKET { $1.add_dimension($3); $$ = $1; }
      | INT { $$ = ast::type(ast::type::_int); }
      | FLOAT { $$ = ast::type(ast::type::_float); }
      | CHAR { $$ = ast::type(ast::type::_char); }
      | VOID { $$ = ast::type(ast::type::_void); }
```

```
      ;

name
      : name LBRACKET expression RBRACKET { $1.add_offset($3); $$ = $1;
}
      | IDENTIFIER { $$ = ast::name($1); }
      ;

nl
      : nl NL
      | NL
      ;

%%

void show_error(const yy::location& l, const std::string &m) {
      std::cerr << "[Error at " << l << "] " << m << std::endl;
}

void yy::cython_parser::error(const location_type& l, const std::string
&m) {
      show_error(l, m);
}

int main(int argc, char** argv) {
      if (argc > 1)
            yyin = std::fopen(argv[1], "r");

      try {
            yy::cython_parser p;
            p.parse();
      } catch (const lexical_error& e) {
            yypop_buffer_state();  // cleans scanner memory
            show_error(e.location(), e.what());
      }

      if (yyin != stdin)
            std::fclose(yyin);
}
```

3. Árvore sintática

Criamos uma estrutura de nodos para o parsing, segue seu código.

```
#ifndef AST_H
#define AST_H

#include <list>
#include <string>

namespace ast {

enum operation {
	plus,
	minus,
	times,
	div,
	exp,
	_and,
	_or,
	gt,
	lt,
	ge,
	le,
	eq,
	ne,
	_not,
	uminus
};

class node {
public:
	node() = default;
};

class block : public node {
public:
	block() = default;
	explicit block(node* line) : node{}, lines{line} {}
	void add_line(node* line) { lines.push_back(line); }

private:
	std::list<node*> lines;
};

class binary_operation : public node {
public:
	binary_operation(operation op, node* left, node* right)
```

```
            : node{}, op{op}, left{left}, right{right} {}

private:
      operation op;
      node* left;
      node* right;
};

class unary_operation : public node {
public:
      unary_operation(operation op, node* operand)
            : node{}, op{op}, operand{operand} {}

private:
      operation op;
      node* operand;
};

class name : public node {
public:
      name() = default;
      explicit name(std::string identifier) : node{},
identifier{identifier} {}
      void add_offset(node* offset) { offsets.push_back(offset); }

private:
      std::string identifier;
      std::list<node*> offsets;
};

class assignment : public node {
public:
      assignment() = default;
      assignment(name variable, node* expression)
            : node{}, variable{variable}, expression{expression} {}

private:
      name variable;
      node* expression;
};

class elif_stmt : public node {
public:
      elif_stmt(node* cond, block elif_block)
            : node{}, cond{cond}, elif_block{elif_block} {}
```

```
private:
	node* cond;
	block elif_block;
};

class if_stmt : public node {
public:
	if_stmt() = default;
	if_stmt(
		node* cond, block if_block, std::list<elif_stmt> elif_stmts,
		block else_block)
		: node{}
		, cond{cond}
		, if_block{if_block}
		, elif_stmts{elif_stmts}
		, else_block{else_block} {}

private:
	node* cond;
	block if_block;
	std::list<elif_stmt> elif_stmts;
	block else_block;
};

class for_stmt : public node {
public:
	for_stmt() = default;
	for_stmt(node* init, node* condition, node* step, block code)
		: node{}, init{init}, condition{condition}, step{step},
code{code} {}

private:
	node* init;
	node* condition;
	node* step;
	block code;
};

class while_stmt : public node {
public:
	while_stmt() = default;
	while_stmt(node* condition, block code)
		: condition{condition}, code{code} {}
```

```
private:
	node* condition;
	block code;
};

class return_stmt : public node {
public:
	return_stmt() = default;
	return_stmt(node* expression) : expression{expression} {}

private:
	node* expression;
};

class int_l : public node {
public:
	explicit int_l(int value) : node{}, value{value} {}

private:
	int value;
};

class float_l : public node {
public:
	explicit float_l(double value) : node{}, value{value} {}

private:
	double value;
};

class string_l : public node {
public:
	explicit string_l(std::string str) : node{}, str{str} {}

private:
	std::string str;
};

class bool_l : public node {
public:
	explicit bool_l(bool b) : node{}, b{b} {}

private:
	bool b;
};
```

```
class type : public node {
public:
	enum _type { _int, _float, _char, _void };

	type() = default;
	explicit type(_type t) : node{}, t{t} {}
	void add_dimension(unsigned int size) {
dimensions.push_back(size); }

private:
	_type t{_void};
	std::list<unsigned int> dimensions;
};

class arg : public node {
public:
	arg() = default;
	arg(std::string identifier, type t, bool reference)
		: node{}, identifier{identifier}, t{t}, reference{reference}
{}

private:
	std::string identifier;
	type t;
	bool reference{false};
};

class declaration : public node {
public:
	declaration(std::string name, type t, node* expression)
		: node{}, name{name}, t{t}, expression{expression} {}

private:
	std::string name;
	type t;
	node* expression;
};

class func : public node {
public:
	func(std::string name, std::list<arg> args, type t, block code)
		: node{}, name{name}, args{args}, t{t}, code{code} {}
	func(std::string name, type t, block code)
		: node{}, name{name}, t{t}, code{code} {}
```

```
private:
	std::string name;
	std::list<arg> args;
	type t;
	block code;
};

class func_call : public node {
public:
	func_call(std::string name, std::list<node*> parameters)
		: node{}, name{name}, parameters{parameters} {}
	func_call(std::string name) : node{}, name{name} {}

private:
	std::string name;
	std::list<node*> parameters;
};

}  // namespace ast

#endif
```

4. Recuperação de Erros

A recuperação de erros implementada retorna o local do erro no código, com a coluna e linha do token. Ele é implementado através da função “show_error” do parser.

5. Exemplos

Foram realizados 5 exemplos curtos para exemplificar o funcionamento do parser. Para realizar a execução dos exemplos, é necessário realizar o comando ***make*** no diretório raiz do código e então executar a seguinte linha de código, ainda no diretório raiz:

```
	./cython examples/nome_do_arquivo_de_exemplo.cy
```

5.1 Fatorial recursivo

Foi criado um código para teste do analisador, definindo uma função de nome factorial. O nome do arquivo de exemplo é “factorial.cy”.

```
def factorial(n: int) -> int begin
	if n == 1 do
		return 1
	else
```

```
            return factorial(n-1) * factorial(n-2)
      end
end
```

5.2 If Elif Else

Segue um código demonstrando a construção do “If elif else”. O nome do arquivo é “if_elif_else.cy”.

```
def main() -> void begin
      i: int = 2
      if i == 1 do
            print("one")
      elif i == 2
            print("two")
      else
            print("i don't know")
      end
end
```

5.3 Máximo de um vetor

Um pequeno código para verificar o uso de matrizes. O nome do arquivo de teste é “find_max.cy”.

```
def find_max(v: int[], size: int) -> int begin
      max = v[0]
      for i: int = 1; i < size; i = i + 1 begin
            if v[i] > max do
                  max = v[i]
            end
      end
      return max
end
```

5.4 Erro léxico

Foi criado um exemplo para causar um erro léxico ao ser executado. O nome do arquivo é “lex_error1.cy”.

```
c: char = '
```

5.5 Erro Sintático

Segue uma pequena modificação para exemplificar o que ocorro quando há um erro sintático. É idêntica ao fatorial mas não possui um “begin”. O nome do arquivo é “syntax_error.cy”.

```
def factorial(n: int) -> int
      if n == 1 do
            return 1
      else
            return factorial(n-1) * factorial(n-2)
      end
end
```

5.6 Referência

Uma exemplificação de passagem de referência por uma função. O arquivo tem nome “reference.cy”.

```
def f(a&: int) -> void:
      a = 1

a = 0
f(a)
print(a)  # should print '1'
```