plgdoc Documentation

Release 1.0

pablo cabeza

September 21, 2014

CONTENTS

CHAPTER

ONE

LEXER AND PARSER

1.1 Structures needed

- **Type symbol table**: A table of types that should consist of:
  - *id*: the name of the type
  - *size*: its width in memory (for allocating)
  - *definition*: some kind of structure defining the type (fields,...)
- **Symbol tables**: The tables of symbols each must contain:
  - *id*: the name
  - *type*: the type of the variable
  - *value*: some kind of pointer or other structure for the value.

  It must also be structured in a stack manner, since there will be scoping to apply to the rules. To accomplish this for each scope there will be a table, and when a new scope is needed a table is added.

  1. An initial table of symbols (globals).
  2. When an scope is added, a table is created and linked to the previous scope table.
  3. When looking for an id, check tables in order from the current scope up.
  4. When leaving an scope remove to top table.
- **String table**: A table for constant strings (global table). This can be use to optimise repeated strings.

1.2 Processing

In the lexer the process to decide if an identifier is a type name or variable name should be:

1. Lookup the *type symbol table*, if the id is there, return a type token of the type found (with a reference to the entry in the table.)
2. Else, return new variable token.

1.3 Operators

Precedence	Operator	Associativtry	Description
	()		function call
	[]	array and unnamed tuple access	
1	.	named tuple access postfix left-right	
	++ –	postfix ++ –	
	type[]	array creation	
	++ –		Prefix ++
	+ -	Sign op	
2	$type$	prefix right-left Casting op	
	$	Typeof operator	
3	* / %		
4	+ -		
5	<< >>		bitwise shift
6	< <= > >=		relational ops
7	== !=	infix left-right	relational ops
8	&		bitwise and
9	^		bitwise xor
10	|		bitwise or
11	&&		bool and
12	||		bool or
13	= += -= *= /= %= <<= >>= &= ^= |=	infix right-left	assignment operators
14	?:	infix (n-ary) right-left	if operator
15	@		loop operator
16	,	infix left-right	comma operator
	‘	apostrophy operator	

CHAPTER

TWO

GRAMMAR FOR THE LANGUAGE

```
(*
        Will there be declarations in the language: int foo(int);
*)

(* Terminal tokens are: *)

IDENTIFIER = ? String from [a-zA-Z][a-zA-Z0-9_]* ? ;
TYPEIDENTIFIER = ? an IDENTIFIER in the type table ? ;
INTEGER = ? [+-]\?[0-9]+ ? ;
FLOAT = ? [+-]\?[0-9]*\.[0-9]+ ? ;
STRING = ? "([\"]|\\")*" ? ;
TYPEMODIFIERS = "const" | "register" | "static"
BASICTYPE = "void" | "int" | "float" | "char" | "bool" ;



(* Types expression rules, an "expression" that returns a type *)

TypeExpr = TYPEMODIFIERS, TypeExprValue


TypeExprValue = TypeName (* Just an IDENTIFIER *)
         | TypeExprValue,"[","]" (* Array Types *)
         | TypeExprValue,"(",TypeList,")" (* Function type *)
         | "(",TypeList,")" (* Tuple without name type *)
         | "(",NamedTypeList,")" (* Tuple with name *)
         | "(",")" (* Empty tuple *) ;

TypeList = TypeList, "," , TypeExpr | TypeExpr ;
NamedTypeList = NamedTypeList, ",", NamedType | NamedType ;

NamedType = TypeExpr,":",IDENTIFIER ;
TypeName = BASICTYPE | TYPEIDENTIFIER ;



(* Definition expressions, everything that "adds": functions, vars, ... *)

Def = VarDef | NamedFunDef | AnomFunDef ;

VarDef = TypeExpr,":",IDENTIFIER,["=",Expr],[",",IdentList] ;
IdentList = IDENTIFIER,["=",Expr] | IDENTIFIER,["=",Expr],",",IdentList ;

NamedFunDef = TypeExpr,IDENTIFIER,"(",NamedTypeList,")","{",StmtList,"}" ;
```

```
AnomFunDef = [ TypeExpr [ ,"(",NamedTypeList")" ] ] ,"{",StmtList,"}" ;



(* Expressions that return a value that can be used *)

ValueExpr = Expr1 ;

Expr1 = Expr1 , '(' , ParameterList , ')' {                    # Block call
                __ = FunctionCallNode(_1,_3) }
          | Expr1 , '[', ValueExpr ,']' {                   # Array or tuple access (rvalue)
                __ = IndexNode(_1,_3,lvalue=_1.lvalue) }
          | Expr1 , '.' ,  IDENTIFIER {                        # Named tuple access (rvalue)
                __ = NameAccessNode(_1,_3,lvalue=_1.lvalue) }
          | Expr1 , ('++' | '--') {                         # postfix in/decrement *)
                __ = PostfixCrementNode(_1,_2) }
          | TypeExpr , '[' , ValueExpr , ']' {                   # Array creation *)
                __ = ArrayCreationNode(_1,_3) }
          | Expr2 {
                __ = _1 }
          ;

Expr2 = ('++' | '--') , Expr2                                (* Prefix in/decrement *)
          | ('+' | '-' ), Expr2                          (* +/- sign operators *)
          | '$' , TypeExpr , '$' , Expr2                 (* type casting operator *)
          | '$' , Expr2                                        (* typeof operator *)
          | Expr3 ;
```

Note: Maybe delete LValue class and have a "flag" in the VariableNode to see if the variable is an RValue or LValue. If a VarNode is built from a variable then it is an LValue, else it is an RValue

CHAPTER

THREE

CODE GENERATION

We are translating bloco code into C language, that should be compiled without problems using an standard C compiler like GCC. All type checks are done in the identifier phase without relying in the compiler, just using the C compiler to translate correct code into object code (which means if a compiler error should be raised, it will be during translation to C).

All generation related code is located in the file *generate.cpp* that implements the method *generate(CodeGenerate& g)* from each subtype of Node.

3.1 CodeGenerate

This class is located in *codegenerate.hpp* completely and is just an auxiliary class that contains the already generated code separating it in head and body sections.

3.2 Auxiliary methods

We have a couple of auxliry methods to help computing offsets or generating repetitive code. Those are:

- **namednodelistgeneratefun()**: this method generates the

3.3 First order functions

The main feature of bloco is first order functions. C doesn't have native support for it, to implement it we have researched various solutions given to this problem and found that the solution adopted by the javascript language (very similar in the sense that it is imperative plus having first order) could be implemented in C with relative ease.

When a function is declared, for example:

```
int: c = 5;
int(int): fun = int (int: a) { r 2*a*c; };
r fun;
```

The differences with an standard C functions would be summarized in:

1. The function in generated *"on the fly"*, interleaved with actual code.
2. The function is passed to a variable of type function, that can be then called.
3. The function has access to variables on the definition scope during execution (closures).

For points number 1. and 2., carefully placing the function definition in a header and referencing it using pointers to function can solve it.

On the other hand, point 3. is harder to implement because it implies that variable should have a longer lifespan than just its defining block, because they can be referenced from an inner defined function, and that function should be able to access symbols from the upper scope in the code.

3.3.1 Frame chaining

The solution we have used to solve this problem is creating a data structure that we call **frame**. A frame is just a chunk of memory where we define local variables of a function, plus a pointer to the *"parent"* frame (the one in the upper scope). Then the structure of a frame is:

```
0       pointer to parent frame
N       function parameters
M       local variables
```

During compilation any reference to symbols is resolved using a pair of number (i,j) where i is the number of scopes it has to go up in the definition chain, and j in the index of that symbol in the frame. The offset of the symbol is computed using when generating the code.

Imagine that (i,j)=(3,2), frameoffset(2)=10 and the symbol is of type *int*, the generated code to reference the symbol would be:

```
( ( int *) ((***(char****) frame) + 10) )[0]
```

Note: The casting *(((char*) frame) + M1)* is very important because, in C *pointer arithmetic*, the offset M1 is added to the pointer *frame* multiplied by the size of it's type, which means we need to use some type of size 1 (char) for this to work properly.

Using this, now each function is translated into a C function that have the form:

```
int __anon_M (void* _frame, int arg1); // M is an unique number
```

And each function will have a prelude that consists on:

1. Allocating the frame using dynamic memory.
2. Assigning the parent frame to this frame.
3. Copying function parameters to the frame.

So the prelude will look like this:

```
void * frame = malloc(N); // Allocate frame
*(void**)frame=_frame; // Save parent frame
((type_arg1*)(((char*) frame) + M1))[0] = arg1; // Copy first parameter
((type_arg2*)(((char*) frame) + M2))[0] = arg2; // Copy second parameter
```

Where **N** is the computed size of the frame and **M1**, **M2**,... are the offsets of their respective function arguments in the frame.

3.3.2 Code and scope binding

Now that the closure problem is solved we need to address the problem of binding a function with its parent frame (the frame of its scope) and calling it. We need a pair *(function, frame)*, that is created when and where a function is defined.

We first define a common C struct for anonymous functions:

```
typedef struct { void* code; void* frame} anon_t;
```

Then the translation of this bloco code:

```
int: c=5;
int(int): fun = int (int: a) { r 2*a*c; };
```

Would look like:

```
int __anon_1 (void* _frame, int a) {
      // Function prologue to create frame
      return 2*A*C;
}

int main() {
      // Create base frame
      C = 5;
      FUN = (anon_t) {__anon_1,frame};
}
```

In the code **A**, **C**, **FUN** are expressions to access their respective symbols in the frame, that are omitted to light the example.

3.3.3 Function calling

At this point calling the function would consist on accessing it's *.code* member, cast it to the right type and call it passing *.frame* as it's first argument and then the rest of parameters. The problem is that it should act as an *expression* because we might need it's return value as part of another expression. This means that we need to *"give it a name"* to access each member. The solution adopted is to generate auxiliary functions that do all this processing.

For example lets translate this code:

```
((int: a) { c = 2*a; })(5); // c is already defined
```

This will translate into 2 functions, *__anon_1* and *__aux_1*, a creation of an object of type *anon_t* and a call to *__aux_1*. The result is:

```
void __anon_1 (void* _frame, int a) {
        // Epilogue of function
        C = 2 * A;
}

void __aux_1 (anon_t f, int f1) {
        ((void (*) ( void*, int )) f.code)( f.frame, f1); // Forward parameters to actual function
}

int main() {
        // main epilogue to allocate base frame
        // ...
        __aux_1 ( (anon_t) { __anon_1 ,frame}, 5 ); // Creation of anon_t and call to __aux_1
}
```

CHAPTER

FOUR

DOCUMENTS

4.1 Links

- Pdf that explains type checking .
- Example for building lex and yacc parser in c++.
- Complete example of c++ parser for flex and bison.