SCANNER

Scanner example based on TypeScript implementation:

{

getText, // returns source input

setText,

getToken, // returns enum which maps to token

getTokenText // returns actual token text

scan, // scans input and returns token

getStartPos, // start position index of token

getTextPos, // current position of input

setTextPos // used to skip ahead of input

}

SYMBOL TABLE

* A data structure, most likely a dictionary which stores identifier values and their attributes, such as; variable name, value, scope, etc.

Struct Table {

Int id;

Int val;

Char\* name;

};

Struct Table\* symbol\_table[MAX\_SYM\_TABLE\_SIZE];

Int table\_size = 0; // current table size

Struct Table\* get\_sym(int id) {

return symbol\_table(id);

}

// most likely used by the scanner to input an identifier

Int add\_sym(char\* name) {

Int i;

For (i = 0; I < table\_size; i++) {

If (strcmp(symbol\_table[i]->name, name) == 0) {

return symbol\_table[i]->id;

}

Struct Table\* item = safe\_malloc(sizeof(struct Table));

Item->id = table\_size;

Item->name = name;

Symbol\_table[table\_size] = item;

return table\_size++;

}

}

* Symbol table is populated by the lexer and used by other parts of the compiler.

*Symbol table after scanning:*

[

{

Id: 0, // first entry in table

Name: “var\_name”,

Val: null // initially null until populated later

},

…

]

* Values can be set at the generator phase

void set\_sym(int id, int val) { // where id is id in symbol table

symbol\_table[id]->val = val;

}

PARSER

The following is taken from the book: *crafting a compiler in C.*

Input: “begin A := BB – 314 + A; end SCANEOF”

void system\_goal() {

program();

match(SCANEOF); // throws error if token not matched expected

}

void program() {

match(BEGIN);

statement\_list();

match(END);

}

Void statement\_list() {

Statement();

While(true) {

Switch(next\_token()) {

case ID:

case READ: // built-in function: read([, ids]);

case WRITE: // built-in function: write();

statement();

break;

default:

return;

}

}

}

void statement() {

token tok = next\_token();

switch (tok) {

case ID:

match(ID);

match(ASSIGNOP);

expression();

match(SEMICOLON);

break;

case READ:

match(READ);

match(LPAREN);

id\_list();

match(RPAREN);

match(SEMICOLON);

break;

case WRITE:

match(WRITE);

match(LPAREN);

expr\_list();

match(RPAREN);

match(SEMICOLON);

break;

default:

// syntax error!

}

}

void id\_list() {

match(ID);

while(next\_token() == COMMA) {

match(COMMA);

match(ID);

}

}

Void expression() {

token t;

primary();

for (t = next\_token(); t == PLUSOP || t == MINUSOP; t = next\_token()) {

add\_op();

primary();

}

}

void expr\_list() {

expression();

while (next\_token() == COMMA) {

match(COMMA);

expression();

}

}

void add\_op() {

token tok = next\_token();

if (tok == PLUSOP || tok == MINUSOP) {

match(tok);

} else {

// syntax error

}

}

void primary() {

token tok = next\_token();

switch (tok) {

case LPAREN:

match(LPAREN);

expression();

match(RPAREN);

break;

case ID:

match(ID);

break;

case INTLITERAL:

match(INTLITERAL);

break;

default:

// syntax error

}

}

* It is quite common for the parser to also build up a parse tree whilst going through the above motions.

ABSTRACT SYNTAX TREE

based on the JS syntax:

const age = 29;

function printName(name) {

console.log(`${name} is ${age} years old`);

}

printName("Ali Issaee");

tree based output in JSON format

{

"type": "Program",

"start": 0,

"end": 117,

"body": [

{

"type": "VariableDeclaration",

"start": 0,

"end": 15,

"declarations": [

{

"type": "VariableDeclarator",

"start": 6,

"end": 14,

"id": {

"type": "Identifier",

"start": 6,

"end": 9,

"name": "age"

},

"init": {

"type": "Literal",

"start": 12,

"end": 14,

"value": 29,

"raw": "29"

}

}

],

"kind": "const"

},

{

"type": "FunctionDeclaration",

"start": 17,

"end": 90,

"id": {

"type": "Identifier",

"start": 26,

"end": 35,

"name": "printName"

},

"expression": false,

"generator": false,

"async": false,

"params": [

{

"type": "Identifier",

"start": 36,

"end": 40,

"name": "name"

}

],

"body": {

"type": "BlockStatement",

"start": 42,

"end": 90,

"body": [

{

"type": "ExpressionStatement",

"start": 45,

"end": 88,

"expression": {

"type": "CallExpression",

"start": 45,

"end": 87,

"callee": {

"type": "MemberExpression",

"start": 45,

"end": 56,

"object": {

"type": "Identifier",

"start": 45,

"end": 52,

"name": "console"

},

"property": {

"type": "Identifier",

"start": 53,

"end": 56,

"name": "log"

},

"computed": false,

"optional": false

},

"arguments": [

{

"type": "TemplateLiteral",

"start": 57,

"end": 86,

"expressions": [

{

"type": "Identifier",

"start": 60,

"end": 64,

"name": "name"

},

{

"type": "Identifier",

"start": 71,

"end": 74,

"name": "age"

}

],

"quasis": [

{

"type": "TemplateElement",

"start": 58,

"end": 58,

"value": {

"raw": "",

"cooked": ""

},

"tail": false

},

{

"type": "TemplateElement",

"start": 65,

"end": 69,

"value": {

"raw": " is ",

"cooked": " is "

},

"tail": false

},

{

"type": "TemplateElement",

"start": 75,

"end": 85,

"value": {

"raw": " years old",

"cooked": " years old"

},

"tail": true

}

]

}

],

"optional": false

}

}

]

}

},

{

"type": "ExpressionStatement",

"start": 92,

"end": 116,

"expression": {

"type": "CallExpression",

"start": 92,

"end": 115,

"callee": {

"type": "Identifier",

"start": 92,

"end": 101,

"name": "printName"

},

"arguments": [

{

"type": "Literal",

"start": 102,

"end": 114,

"value": "Ali Issaee",

"raw": "\"Ali Issaee\""

}

],

"optional": false

}

}

],

"sourceType": "module"

}

GENERATOR

*Based on tiny-compiler (written in C), on github.*

* Recursively calls itself building up a string of output based on the source code type.

Switch (node->type) {

case IDENTIFIER:

gen(READ); // this eventually gets added to the array

break;

// more cases

case NUM\_TYPE:

gen(PUSH);

gen(node->val);

break;

case SET\_TYPE:

gen(WRITE);

gen(node->op1->val); // where this produces the var name

break;

…

}

enum Types = {PUSH, POP, READ, WRITE, …};

using the above switch statement – the input: *cath1 = 7;* would result in the following output:

*[0, 7, 3, 0]; // the last zero is the var pos in the symbol table*

We can now use this array to produce some asm code like:

PUSH 7

WRITE cath1

RET

*Using output from above example: [0, 7, 3, 0];*

While (*less than output array*) {

Switch (*current index in array*) {

case PUSH:

printf(“PUSH %i\n”, output[++i]); // std output

break;

case WRITE:

printf(“WRITE %s\n”, **get\_sym**(value)->name);

break;

}

}

printf(“RET\n”);

* get\_sym returns the name of the variable from the symbol table.