

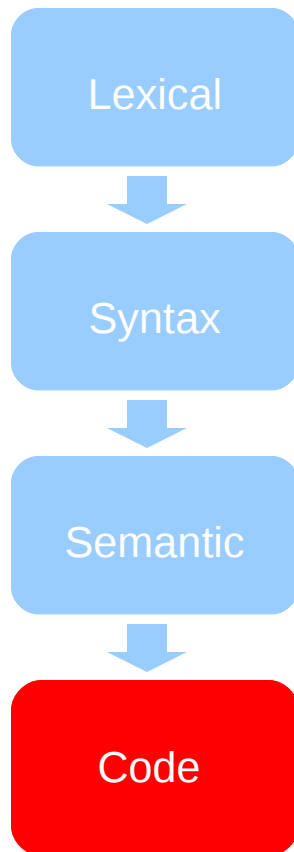
Experiment in Compiler Construction Code Generation (1)

**School of Information and
Communication Technology
Hanoi University of Science and
Technology**

Overview

- Code generation overview
- Stack calculator
 - Stack calculator's memory
 - Instruction set
- Additional changes in symbol table
 - Variables
 - Parameters
 - Program, functions, and procedures

What is code generation?

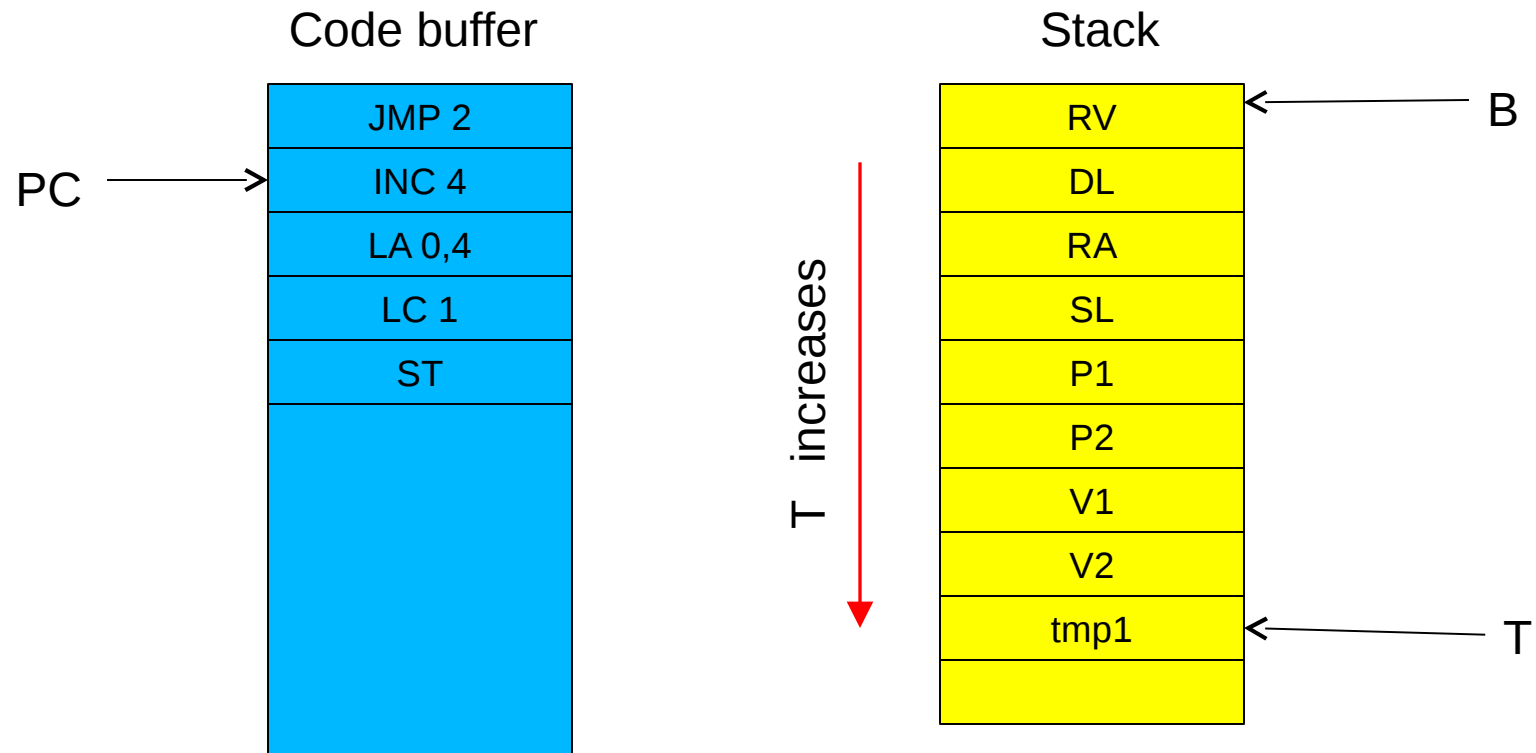


- Code generation is the phase that generates a sequence of target machine instructions corresponding to the source program's grammar.
- Program's grammar is checked and built by the syntax analyzer (parser)
- Target machine instructions are specified in execution model of target machine

Stack calculator

- Stack calculator is a computing system
 - Using stack to store intermediate results during computation process.
 - Simple organization
 - Simple instruction set
- Stack calculator consists of 2 memory areas
 - Code buffer: containing execution code corresponding to source program
 - Stack: storing intermediate results

Stack calculator



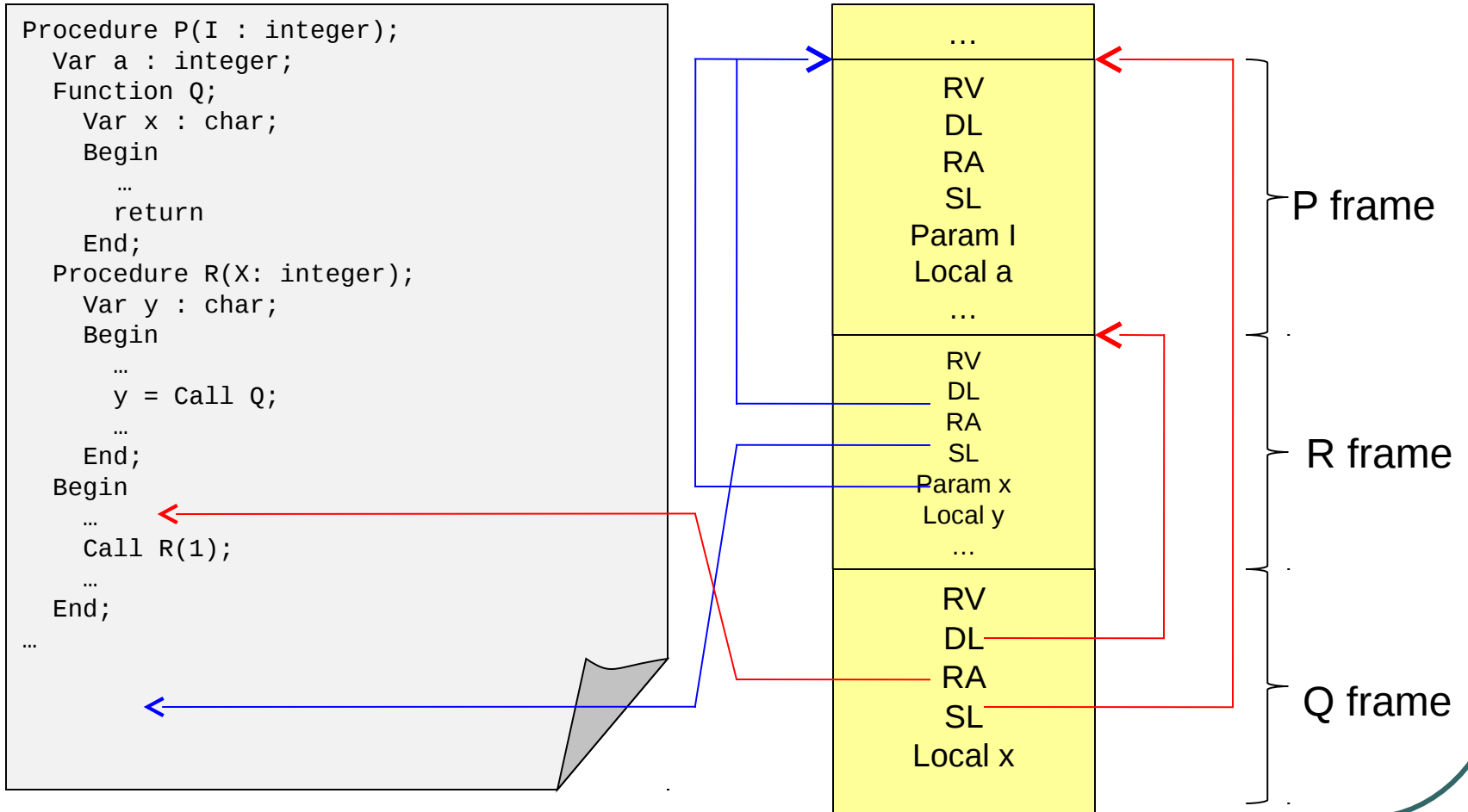
Stack calculator

- Registers
 - PC (program counter): pointing to currently being executed instruction on Code buffer
 - B (base): pointing to the base address of data area of active block on Stack. Local variables are accessed via B
 - T (top): pointing to Stack's top element

Stack calculator

- **Activation record / Stack frame**
 - Is the memory area allocated to every function, procedure and the main program when it is activated (becoming active block)
 - Storing parameters' values
 - Storing local variables's values
 - Other information
 - Return value – RV
 - Dynamic link – DL
 - Return address – RA
 - Static link – SL
 - A function/procedure may have several Stack frames on Stack

Stack calculator



Stack calculator

- RV (return value): stores return value of a function
- DL (dynamic link): is the base address of caller's Stack frame. DL is used to recover caller's context when the callee ends.
- RA (return address): address of caller's instruction that would be executed when callee ends.
- SL (static link): base address of outer's Stack frame. SL is useful when we track non-local variables.

Stack calculator

- Instruction set

op	p	q
----	---	---

LA	Load Address	$t := t + 1; s[t] := \text{base}(p) + q;$
LV	Load Value	$t := t + 1; s[t] := s[\text{base}(p) + q];$
LC	Load Constant	$t := t + 1; s[t] := q;$
LI	Load Indirect	$s[t] := s[s[t]];$
INT	Increment T	$t := t + q;$
DCT	Decrement T	$t := t - q;$

Stack calculator

- Instruction set

op	p	q
----	---	---

J	Jump	pc:=q;
FJ	False Jump	if s[t]=0 then pc:=q; t:=t-1;
HL	Halt	Halt
ST	Store	s[s[t-1]]:=s[t]; t:=t-2;
CALL	Call	s[t+2]:=b; s[t+3]:=pc; s[t+4]:=base(p); b:=t+1; pc:=q;
EP	Exit Procedure	t:=b-1; pc:=s[b+2]; b:=s[b+1];
EF	Exit Function	t:=b; pc:=s[b+2]; b:=s[b+1];

Stack calculator

- Instruction set

op	p	q
----	---	---

RC	Read Character	read one character into s[s[t]]; t:=t-1;
RI	Read Integer	read integer to s[s[t]]; t:=t-1;
WRC	Write Character	write one character from s[t]; t:=t-1;
WRI	Write Integer	write integer from s[t]; t:=t-1;
WLN	New Line	CR & LF

Stack calculator

- Instruction set

op	p	q
----	---	---

AD	Add	$t := t - 1; s[t] := s[t] + s[t + 1];$
SB	Subtract	$t := t - 1; s[t] := s[t] - s[t + 1];$
ML	Multiply	$t := t - 1; s[t] := s[t] * s[t + 1];$
DV	Divide	$t := t - 1; s[t] := s[t] / s[t + 1];$
NEG	Negative	$s[t] := -s[t];$
CV	Copy Top of Stack	$s[t + 1] := s[t]; t := t + 1;$

Stack calculator

- Instruction set

op	p	q
----	---	---

EQ	Equal	$t := t - 1$; if $s[t] = s[t+1]$ then $s[t] := 1$ else $s[t] := 0$;
NE	Not Equal	$t := t - 1$; if $s[t] \neq s[t+1]$ then $s[t] := 1$ else $s[t] := 0$;
GT	Greater Than	$t := t - 1$; if $s[t] > s[t+1]$ then $s[t] := 1$ else $s[t] := 0$;
LT	Less Than	$t := t - 1$; if $s[t] < s[t+1]$ then $s[t] := 1$ else $s[t] := 0$;
GE	Greater Equal or	$t := t - 1$; if $s[t] \geq s[t+1]$ then $s[t] := 1$ else $s[t] := 0$;
LE	Less Equal or	$t := t - 1$; if $s[t] \leq s[t+1]$ then $s[t] := 1$ else $s[t] := 0$;

Changes in symbol table

- Variable's new attributes
 - localOffset: variable's location on local frame.
 - scope
- Parameter's new attributes
 - localOffset: parameter's location on local frame.
 - scope
- Program/function/procedure's new attributes
 - codeAddress: address of first instruction on Code buffer
 - frameSize: size of corresponding Stack frame
 - paramCount: number of parameters

Changes in symbol table

- Variable's new attributes
 - Scope
 - localOffset: location in local frame (its distance to local frame's base)

```
struct VariableAttributes_ {  
    Type *type;  
    struct Scope_ *scope;  
    int localOffset;  
};
```


Changes in symbol table

- Parameter's new attributes
 - Scope
 - localOffset

```
struct ParameterAttributes_ {  
    enum ParamKind kind;  
    Type* type;  
    struct Scope_ *scope;  
    int localOffset;  
};
```

Changes in symbol table

- Scope's new attribute
 - frameSize

```
struct Scope_ {  
    ObjectNode *objList;  
    Object *owner;  
    struct Scope_ *outer;  
    int frameSize;  
};
```

Changes in symbol table

- Function's new attributes
 - codeAddress
 - paramCount

```
struct FunctionAttributes_ {  
    struct ObjectNode_ *paramList;  
    Type* returnType;  
    struct Scope_ *scope;  
  
    int paramCount;  
    CodeAddress codeAddress;  
};
```

Changes in symbol table

- Procedure's new attributes
 - codeAddress
 - paramCount

```
struct ProcedureAttributes_ {  
    struct ObjectNode_ *paramList;  
    struct Scope_* scope;  
  
    int paramCount;  
    CodeAddress codeAddress;  
};
```

Changes in symbol table

- Program's new attribute
 - codeAddress

```
struct ProgramAttributes_ {  
    struct Scope_ *scope;  
    CodeAddress codeAddress;  
};
```

Assignments

- Implement following function in *syntab.c*

```
int sizeofType(Type* type);  
void declareObject(Object* obj);
```
- Note: for simplicity, each integer/char occupies one word (4 bytes) in Stack
- Order of words in a local frame is as following:
 - 0: RV
 - 1: DL
 - 2: RA
 - 3: SL
 - $4 \rightarrow (4+k)$: for k parameters
 - $(4+k+1) \rightarrow (4+k+n)$: for local variables

Experiment in Compiler Construction

Code generation (2)

Nguyen Huu Duc

Department of Information Systems
Faculty of Information Technology
Hanoi University of Technology

Overview

- kplrun utility
- Overview of instructions.*, codegen.*
- Generate code for (no subprogram/array)
 - ASSIGN (substitute) statement
 - IF statement
 - WHILE statement
 - FOR statement
 - CONDITION
 - EXPRESSION

kplrun

- Interpreter for Stack calculator. Syntax:

```
$ kplrun <source> [-s=stack-size] [-c=code-size] [-debug] [-dump]
```

- Options:

- s: define Stack size

- c: define maximum size of source program

- dump: output generated instruction code to standard output

- debug: debugging mode

kplrun

- Options in debugging mode
 - a: corresponding absolute address of a Stack location (level, offset)
 - v: value stored in a Stack location (level, offset)
 - t: value stored in Stack's top
 - c: exit debugging mode

Instructions.c

```
enum OpCode {  
    OP_LA,    // Load Address:  
    OP_LV,    // Load Value:  
    OP_LC,    // load Constant  
    OP_LI,    // Load Indirect  
    OP_INT,   // Increment t  
    OP_DCT,   // Decrement t  
    OP_J,     // Jump  
    OP_FJ,    // False Jump  
    OP_HL,    // Halt  
    OP_ST,    // Store  
    OP_CALL,  // Call  
    OP_EP,    // Exit Procedure  
    OP_EF,    // Exit Function
```

```
    OP_RC,    // Read Char  
    OP_RI,    // Read Integer  
    OP_WRC,   // Write Char  
    OP_WRI,   // Write Int  
    OP_WLN,   // WriteLN  
    OP_AD,    // Add  
    OP_SB,    // Subtract  
    OP_ML,    // Multiple  
    OP_DV,    // Divide  
    OP_NEG,   // Negative  
    OP_CV,    // Copy Top  
    OP_EQ,    // Equal  
    OP_NE,    // Not Equal  
    OP_GT,    // Greater  
    OP_LT,    // Less  
    OP_GE,    // Greater or Equal  
    OP_LE,    // Less or Equal  
  
    OP_BP     // Break point.
```

```
};
```

Instructions.c

```
struct Instruction_ {  
    enum OpCode op;  
    WORD p;  
    WORD q;  
};
```

```
struct CodeBlock_  
{  
    Instruction* code;  
    int codeSize;  
    int maxSize;  
};
```

```
CodeBlock* createCodeBlock(int maxSize);  
void freeCodeBlock(CodeBlock* codeBlock);  
void printInstruction(Instruction* instruction);  
void printCodeBlock(CodeBlock* codeBlock);
```

```
void loadCode(CodeBlock* codeBlock, FILE* f);  
void saveCode(CodeBlock* codeBlock, FILE* f);
```

```
int emitLA(CodeBlock* codeBlock, WORD p, WORD q);  
int emitLV(CodeBlock* codeBlock, WORD p, WORD q);  
int emitLC(CodeBlock* codeBlock, WORD q);
```

```
...  
int emitLT(CodeBlock* codeBlock);  
int emitGE(CodeBlock* codeBlock);  
int emitLE(CodeBlock* codeBlock);
```

```
int emitBP(CodeBlock* codeBlock);
```

codegen.c

```
void initCodeBuffer(void);
void printCodeBuffer(void);
void cleanCodeBuffer(void);
int serialize(char* fileName);

int genLA(int level, int offset);
int genLV(int level, int offset);
int genLC(WORD constant);
...
int genLT(void);
int emitGE(void);
int emitLE(void);
```

Generate code for ASSIGN statement

$v := \text{exp}$

```
<code of l-value v> // load address of v  
<code of exp>       // load value of of exp  
ST
```

Generate code for IF statement

If <cond> Then statement;

```
<code of cond>      // load value of condition
FJ L
<code of statement>
L:
...
```

If <cond> Then st1 Else st2;

```
<code of cond>      // load value of condition
FJ L1
<code of st1>
J L2
L1:
  <code of st2>
L2:
...
```

Generate code for WHILE statement

While <cond> Do statement

```
L1:
    <code of cond> // load value of condition
    FJ L2
    <code of statement>
    J L1
L2:
    ...
```


Generate code for FOR statement

For v := exp1 to exp2 do statement

```
<code of l-value v>
CV    // copy top of stack - duplicate address of v
<code of exp1>
ST    // store original value of v
L1:
CV
LI    // get value of v
<code of exp2>
LE
FJ L2
<code of statement>
CV;CV;LI;LC 1;AD;ST; // increase v's value by 1
J L1
L2:
DCT 1
...
```

Assignments

- Complete following function in *codegen.c*
 - `genVariableAddress(Object* var)`
// push address of a variable to Stack's top
 - `genVariableValue(Object* var)`
// push value of a variable to Stack's top

Note: non-local variable temporarily
exclusive

Assignments

- Complete following functions in *parser.c*
 - Generate code for a variable l-value
 - Generate code for statements: Assign, If, While, For
 - Generate code for Condition
 - Generate code for Expression

Experiment in Compiler Construction

Code generation (3)

Nguyen Huu Duc

Department of Information Systems
Faculty of Information Technology
Hanoi University of Technology

Overview

- Generate code for variable's address/value (non-local inclusive)
- Generate code for parameter's address/value (non-local inclusive)
- Generate code for address of function's return value
- Generate code for calling function/procedure
 - Generate code for arguments
- Treatment of array

Generate code for VARIABLE's address and value

- When generate code for a variable's address/value, pay attention to its scope
 - Local variable: track in active Stack frame
 - Non-local variable: track static links and depth of tracking equals depth from current scope to variable's scope

computeNestedLevel(Scope* scope)

Generate code for PARAMETER's address

- Dosage: when **LValue** is a parameter
- As variable, pay attention to its scope
- Call by value: push to top of Stack parameter's address.
- Call by reference: push to top of Stack parameter's value

Generate code for PARAMETER's value

- Dosage: when compute value of **Factor**
- As variable, pay attention to its scope
- Call by value: push to Stack parameter's value
- Call by reference: push to Stack the value located at the address which is parameter's value

Generate code for address of FUNCTION's return value

- Offset = 0.
- Level = depth from current scope to function's scope

Generate code for CALLING function/procedure

- Dosage
 - Calling a function: when generate code for **factor**
 - Calling a procedure: when generate code for **callst** statement.
- Preparation: identify values of parameters
 - Increase value of T by 4 (omit RV, DL, RA, SL)
 - Generate code for k arguments
 - Decrease value of T by 4 + k
 - Generate code for CALL statement

Generate code for CALL(p,q) statement

```
CALL (p, q) s[t+2]:=b; // store dynamic link  
             s[t+3]:=pc; // store return address  
             s[t+4]:=base(p); // store static link  
             b:=t+1; // new base, new return value address  
             pc:=q; // jump to new instruction
```

CALL (p, q) to a function/procedure A require 2 parameters

p: Depth of CALL statement

= depth of A's outer

= depth from current scope to scope of A's outer

p tells A's static link

q: Address of new instruction code

Operation of stack calculator when a CALL(p, q) instruction is performed

1. **pc** changes to codeAddress (beginning address) of called sub-program $/* \text{pc} = p */$
2. Increase **pc** by 1 $/* \text{pc} ++ */$
3. First code instruction would be Jump instruction **J** to omit code instruction of local declaration in **code buffer**.
4. Next statement would be **INT** to increase **T** exactly by size of frame to omit Stack area corresponding to local parameters and variables.

Operation of stack calculator when a CALL(p, q) instruction is performed

5. Execute next instructions and Stack would changes correspondingly.
6. Ending
 1. A procedure (instruction **EP**): release active frame and set **T** to previous frame's top.
 2. Function (lệnh **EF**): release active frame, except return value at **offset 0**, set **T** to **offset 0**.

Treatment of ARRAY

- An array that is declared like

$A : \text{array}(.n_1.) \text{ of } \dots \text{ of array}(.n_k.) \text{ of integer/char}$
would occupies $n_1 * \dots * n_k$ word in Stack frame

- Element $A(.i_1.) \dots (.i_k.)$ is located at address

$$= A + (i_1 - 1) * n_2 * \dots * n_k$$

$$+ (i_2 - 1) * n_3 * \dots * n_k$$

...

$$+ (i_{k-1} - 1) * n_k$$

$$+ (i_k - 1)$$

- This address is accumulated when compiling indexes

Assignments

- Complete functions in *codegen.c*

```
int computeNestedLevel(Scope* scope);  
void genVariableAddress(Object* var)  
void genVariableValue(Object* var)  
void genParameterAddress(Object* param)  
void genParameterValue(Object* param)  
void genReturnValueAddress(Object* func)  
void genReturnValueValue(Object* func)  
void genProcedureCall(Object* proc)  
void genFunctionCall(Object* func)
```

Assignments

- Make changes to *parser.c*

```
Type* compileLValue(void);
```

```
void compileCallSt(void);
```

```
Type* compileFactor(void);
```

```
Type* compileIndexes(Type* arrayType);
```