HEIG-VD – 서울대 공대 SU Computer Science 2016
The Art of Compiler Construction

Course Project

```
type<n>    NUMBER
type<str>  ident string IDENT STRING
type<t>    type
type<idl>  identl vardecl
type<opc>  condition
type<bpr>  IF WHILE

%

rogram     :                                  { stack = init_stack(NULL); symtab = init_symtab(stack, NULL); }
             decll                            { cb = init_codeblock("");
                                                stack = init_stack(stack); symtab = init_symtab(stack, symtab);
                                                rettype = tVoid;
                                              }
             stmtblock                        { stack = stack->uplink; symtab = symtab->parent;
                                                dump_codeblock(cb); save_codeblock(cb, fn_pfx);
                                              }
           ;

ecll       : %empty
           | decll vardecl ';'                { free_idlist($vardecl); }
           | decll fundecl
           ;

ardecl     : type identl                      {
                                                IDlist *l = $identl;
                                                if ($type != tInteger) {
                                                  yyerror("Invalid type for variable declaration.");
                                                  YYABORT;
                                                }
                                                while (1) {
                                                  if (insert_symbol(symtab, l->id, $type) == NULL) {
                                                    char *error = NULL;
                                                    asprintf(&error, "Duplicated identifier '%s'.", l->id);
```

The su (Summer University) Programming Language and Runtime Environment

Components

- **suPL**
  - a simple, C-like language
  - definition given as EBNF
- **suplc**
  - suPL compiler
  - written using Flex/Bison
  - **this is your job**
- **suVM**
  - a stack-based “computer” that can execute compiled suPL code
  - provided

Extended Backus-Naur Form

Extended Backus-Naur Form

- A formal notation to write down programming languages (context-free grammars)
  - meta symbols: “=” (definition), “,” (concatenation), “;” (end of production)

```
module   =   "module", ident, ";",
             { [typeDecl], [varDecl], [funcDecl] },
             "begin", stmtSeq, "end", ident, ".";
```

  - we use no symbol for concatenation, and “.” to indicate the end of the production

```
module   =   "module" ident ";"
             { [typeDecl] [varDecl] [funcDecl] }
             "begin" stmtSeq "end" ident ".".
```

Extended Backus-Naur Form

- Notation: Extended Backus-Naur Form

<table>
<tr><th>Notation</th><th>Usage</th><th>Example</th></tr>
<tr><td>=</td><td>definition</td><td><code>letter = "A".."Z".</code></td></tr>
<tr><td>.</td><td>termination</td><td><code>letter = "A".."Z".</code></td></tr>
<tr><td>|</td><td>alternation</td><td><code>letter = "A".."Z" | "a".."z".</code></td></tr>
<tr><td>[…]</td><td>option</td><td><code>number = ["-"] digit.</code></td></tr>
<tr><td>{ … }</td><td>repetition (≥ 0)</td><td><code>number = ["-"] digit {digit}.</code></td></tr>
<tr><td>(…)</td><td>grouping</td><td><code>factor = [unaryOp] (ident | number).</code></td></tr>
<tr><td>"…", '…'</td><td>terminal symbol</td><td><code>"module", '"'</code></td></tr>
</table>

The su Programming Language

The su Programming Language

- Simple C-based programming language
  - one data type: 32-bit signed integer
  - function calls (reentrant)
  - expression support
    - binary integer operations: +, -, *, /, %, ^
    - negation (“-a”) not supported
    - parentheses
  - control flow constructs
    - if – else
    - while
    - call – return
  - basic I/O
    - read/write
    - print string

suPL Example

```
int a, b;

int add(int p1, p2) {
  return p1 + p2;
}

void count(int N) {
  print “Counting to ”;
  write N;

  int i;

  i = 0;
  while (i < N) {
    write i;
    i = i + 1;
  }
}

{
  int n;

  print “Enter n: ”;
  read n;
  write add(1, n);
  count(n);
}
```

```
$ suplc test.su
$ suvm test.sux
Enter n: 5
6
Counting to 5
0
1
2
3
4
$
```

EBNF of suPL

```
program              =   decll stmtblock.


decll                =   { vardecl ‘;’ | fundecl }.

vardecl              =   type ident { ‘,’ ident }.

type                 =   “integer” | “void”.

fundecl              =   type ident ‘(’ [ vardecl ] ‘)’ stmtblock.


stmtblock            =   ‘{’ { stmt } ‘}’.

stmt                 =   vardecl ‘;’ | assign | if | while | call ‘;’ | return |
                         read | write | print.
assign               =   ident ‘=’ expression.

if                   =   “if” ‘(’ condition ‘)’ stmtblock [ “else” stmtblock ].

while                =   “while” ‘(’ condition ‘)’ stmtblock.

call                 =   ident ‘(’ [ expression { ‘,’ expression } ] ‘)’.

return               =   “return” [ expression ] ‘;’.
```

EBNF of suPL

```
read          =   “read” ident ‘;’.

write         =   “write” expression ‘;’.

print         =   “print” string ‘;’.


expression    =   number | ident |
                  expression ‘+’ expression | expression ‘-’ expression |
                  expression ‘*’ expression | expression ‘/’ expression |
                  expression ‘%’ expression | expression ‘^’ expression |
                  ‘(’ expression ‘)’ | call.
condition     =   expression “==” expression |
                  expression “<=” expression |
                  expression ‘<’ expression.


number        =   DIGIT { DIGIT }.

ident         =   ALPHA { (ALPHA | DIGIT) }.

string        =   ‘“’ { printable ASCII | “\t” | “\n” | ‘\”’ | “\\” } ‘”’.
```

The suPL Compiler

- Flex/Bison-based compiler
  - you write the Flex/Bison files
  - helper functions and data structures provided
- Code generation on the fly
- Code format: binary

The su Virtual Machine

- Simulates a stack-based processor
  - a = b + 7

```
load b       load ‘b’ and push onto operand stack
push 7       push 7 onto operand stack
add          pop twice ($1, $2), compute $2+$1, push result
store a      pop value on top of stack, store into ‘a’
```

- Separate stack for variables & globals

Project Phase 1

suPL Scanner

suPL Scanner

- Prerequisites
  - Linux/POSIX system (MacOS should work)
    or
    VirtualBox development VM
- Your task: write a scanner (supl.lex) that correctly tokenizes suPL
- Hints and resources:
  - Flex manual: http://flex.sourceforge.net/manual/
  - identify each keyword separately (this will be necessary for the second phase)