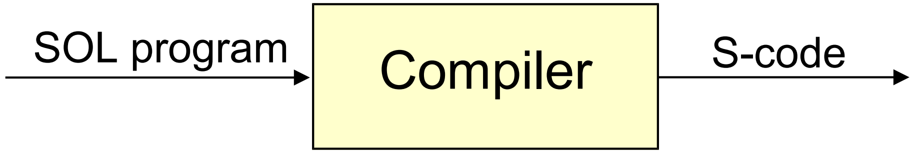
Code Generation

```
func prog(a: int; b: string;): int
  type T1: …, T2: …, Tn: …;
  var v1: …, v2: …, vm: …;
  const c1: … = …, c2: … = …, cp: … = …;

  func f1(…): T1
    type …;
    var …;
    const …;

    func f2(…): T2
      ...
    begin f2
      ...
    end f2

  begin f1
    ...
  end f1

  ...

begin prog
  ...
end prog
```




```
SCODE size

PUSH num-formals num-locals -1      static chain
  GOTO ^prog
POP
HALT

⟨prog⟩

⟨f1⟩

⟨f2⟩

...
```

Code Generation (ii)

- **Design choices:**
  - Program treated as a function (callable from other functions)
  - Program parameters = local variables
  - Code directly addressable (without labels)
  - Address of S-code statement = position of statement within generated code
  - Object descriptors allocated in the order they are declared (formals, var, const)
  - Objects (param, var, const) treated uniformly by abstract machine (S-machine)
  - Object identification (param, var, const): *offset-env, oid*

Variable Definition

- Two sorts of objects $\begin{cases}\text{embedded (atomic)} \\ \text{on stack (struct, vector)}\end{cases}$

```
var
  c: char;
  i: int;
  x: real;
  s: string;
  b: bool;
  r: struct(a: char; b: string;);
  v1: vector [5] of int;
  v2: vector [100] of struct(a: int; b: char;);
```

⇨

```
NEW  |char|
NEW  |int|
NEW  |float|
NEW  |string|
NEW  |char|
NEWS <|char|+|string|>
NEWS <5*|int|>
NEWS <100*(|int|+|char|)>
```

- **Notes:**
  - Atomic object (embedded instance): `NEW` *object-size*
  - Struct/vector (instance on stack): `NEWS` *object-size*

Constant Definition

- Generation of `NEW`, `NEWS` (as for variables)
- Generation of code for value instantiation (after all `NEW`…)

```
const
     MAX: int = 100;
     name: T2 = "alpha";
     PAIR: struct(a: int; b: char;) = struct(25, 'c');
     VECT: vector [5] of real = vector(2.0, 3.12, 4.67, 1.1, 23.0);
     MAT: vector [2] of vector [5] of real = vector(VECT, vector(x, y, z, 10.0, x+y+z));
```

```
NEW  |int|
NEW  |string|
NEWS <|int|+|char|>
NEWS <5*|float|>
NEWS <2*5*|float|>
<instantiation of MAX>
<instantiation of name>
<instantiation of PAIR>
<instantiation of VECT>
<instantiation of MAT>
```

Reference to Atomic Constants

1. Character constant	`x = 'c';`	⇨	`LDC 'c'`
2. Integer constant	`y = 25;`	⇨	`LDI 25`
3. Real constant	`z = 3.14;`	⇨	`LDR 3.14`
4. String constant	`s = "alpha";`	⇨	`LDS "alpha"`
5. Boolean constant	`b = true;`	⇨	`LDC '1'`

- **Note:**
  - Boolean values: **`true`**, **`false`** → surrogated by characters: '1', '0'

Reference to Identifiers

1. Local object

```
x = y + 1;    ⇨   LOD 0 ^y
r1 = r2;      ⇨   LOD 0 ^r2
```

```
var
  x, y: real;
  r1, r2: struct(a: int; b: string;);
  v: vector [10] of real;
```

2. Nonlocal object

```
x = z + 1;    ⇨   LOD 2 ^z
v = w;        ⇨   LOD 3 ^w
```

- **Note:**
  - Arguments of `LOD` = *env-offset, oid* (object identifier)

Instance Constructors: `struct`

- Generation of code of expressions of structure fields + chaining statement (`CAT`)

```
const  r: struct(a: int; b: vector [10] of string; c: real;) = struct(i+j, v, f(x));
```

```
<i+j>
LOD 0 ^v
<f(x)>
CAT 3 <|int|+(10*|string|)+|float|>
```

- **Note:**
  - Arguments of `CAT` = *num-fields, struct-size*

Instance Constructors: `vector`

- Generation of code of expressions of vector elements + chaining statement (`CAT`)

```
val  v: vector [3] of struct(a: int; b: real;) = vector(struct(i+j, x), struct(7, 3.14), struct(i-j, f(x)));
```

```
<i+j>
LOD 0 ^x
CAT 2 <|int|+|float|>
LDI 7
LDR 3.14
CAT 2 <|int|+|float|>
<i-j>
<f(x)>
CAT 2 <|int|+|float|>
CAT 3 <3*(|int|+|float|)>
```

- **Note:**
  - Arguments of `CAT` = *num-elements, vector-size*

Reference by Fielding

- Loading of address of struct instance (`LDA`) +
  loading of address of struct field (`FDA`) +
  indirect load (`EIL`, `SIL`)

```
var i: int;
    r: struct(a: real; b: int;);

write (i + r.b);
```

```
LDA 0 ^r
FDA |float|
EIL |int|
```

```
var r1: struct(a: real; r2: struct(c: char; d: vector [10] of string;));

write r1.r2.d;
```

```
LDA 0 ^r1
FDA |float|
FDA |char|
SIL <10*|string|>
```

- **Notes:**
  - Argument of `LDA` = *env-offset, oid*
  - Argument of `FDA` (field address) = *field-offset*
  - Argument of `EIL`, `SIL` (indirect load, either embedded or on stack) = *field-size*

Reference by Indexing

- Loading of address of vector instance (`LDA`) +
  computation of index value +
  loading of address of vector element (`IXA`) +
  indirect load (`EIL`, `SIL`)

```
var i,j: int;
    v: vector [10] of int;

write v[i+j];
```

```
LDA 0 ^v
<i+j>
IXA |int|
EIL |int|
```

```
var i,j: int;
    v: vector [10] of vector [20] of int;

write v[i-j][i+j];
```

```
LDA 0 ^v
<i-j>
IXA <20*|int|>
<i+j>
IXA |int|
EIL |int|
```

- **Notes:**
  - Argument of `LDA` (load address) = *env-offset, oid*
  - Argument of `IXA` (indexed address) = *elem-size*
  - Argument of `EIL`, `SIL` (indirect load) = *elem-size*

Assignment of Identifiers

- Computation of assignment expression + store (`STO`)

1. Assignment of local object (var, param):

`x = `*expr*`;` $\Rightarrow$

```
⟨expr⟩
STO 0 ^x
```

2. Assignment of nonlocal object (var, param):

`y = `*expr*`;` $\Rightarrow$

```
⟨expr⟩
STO 2 ^y
```

- **Note:**
  - Argument of `STO` = *env-offset, oid* (of assigned object)

Assignment of Fielding

- Computation of address of field to assign + computation of assignment expression + indirect store (IST)

```
var i: int;
    r: struct(a: real; b: int;);

r.b = i+j;
```

```
LDA ^r
FDA |float|
<i+j>
IST
```

```
var r1: struct(a: real; r2: struct(c: char; d: int;));

r1.r2.d = i+j;
```

```
LDA 0 ^r1
FDA |float|
FDA |char|
<i+j>
IST
```

- **Note:**
  - IST = operator without arguments

Assignment of Indexing

- Computation of address of element to assign + computation of assignment expression + indirect store (`IST`)

```
var i,j: int;
    v: vector [10] of int;

v[i+j] = i*j;
```

```
LDA ^v
<i+j>
IXA |int|
<i*j>
IST
```

```
var i,j: int;
    v: vector [10] of vector [20] of int;

v[i-j][i+j] = i*j;
```

```
LDA 0 ^v
<i-j>
IXA <20*|int|>
<i+j>
IXA |int|
<i*j>
IST
```

- **Note:**
  - Uniformity of assignment of fielding and indexing (`IST`)

Logical Operations (`and`, `or`)

logic-expr $\rightarrow$ *expr*$_1$ *expr*$_2$ ⇨

`and`

```
⟨ expr1 ⟩
JMF  offset
⟨ expr2 ⟩
JMP  2
LDC  '0'
```

`or`

```
⟨ expr1 ⟩
JMF  3
LDC  '1'
JMP  exit
⟨ expr2 ⟩
```

```
var
  a, b, c: bool;
  …
  a = (b and c) or g;
```

⇨

```
LOD  0  ^b
JMF  3
LOD  0  ^c
JMP  2
LDC  '0'
JMF  3
LDC  '1'
JMP  2
LOD  2  ^g
STO  0  ^a
```

- **Notes:**
  - Short circuit evaluation
  - `JMP` = unconditional jump
  - `JMF` = conditional jump (to false)
  - Argument of `JMP`, `JMF` = extent of jump (offset) $\begin{cases} exit = |\langle expr_2 \rangle| + 1 \\ offset = |\langle expr_2 \rangle| + 2 \end{cases}$

Relational Operations: ==, !=

$rel\text{-}expr \rightarrow expr_1\ \ expr_2$ ⇒

==	!=
⟨ $expr_1$ ⟩	⟨ $expr_1$ ⟩
⟨ $expr_2$ ⟩	⟨ $expr_2$ ⟩
`EQU`	`NEQ`

```
var
  i, j: int;
  b: bool;

  b = (i == j or j != k);
```

⇒

```
LOD 0 ^i
LOD 0 ^j
EQU
JMF 3
LDC '1'
JMP 4
LOD 0 ^j
LOD 2 ^k
NEQ
STO 0 ^b
```

- **Note:**
  - `EQU`, `NEQ`: polymorphic for all sorts of objects

Relational Operations: >, >=, <, <=

	>	>=	<	<=
char	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `CGT`	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `CGE`	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `CLT`	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `CLE`
int	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `IGT`	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `IGE`	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `ILT`	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `ILE`
real	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `RGT`	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `RGE`	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `RLT`	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `RLE`
string	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `SGT`	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `SGE`	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `SLT`	$\langle\, expr_1 \rangle$ $\langle\, expr_2 \rangle$ `SLE`

$$\textit{rel-expr} \rightarrow expr_1 \;\; expr_2 \quad \Rightarrow$$

```
var
  i, j: int;
  x, y: real
  b: bool;

  b = (i > j or x < y);
```

⇒

```
LOD  0  ^i
LOD  0  ^j
IGT
JMF  3
LDC  '1'
JMP  4
LOD  0  ^x
LOD  0  ^y
RLT
STO  0  ^b
```

Relational Operations: `in`

$$\textit{rel-expr} \rightarrow \textit{expr}_1 \ \ \textit{expr}_2 \quad \Rightarrow \quad \begin{array}{l} \langle \textit{expr}_1 \rangle \\ \langle \textit{expr}_2 \rangle \\ \texttt{IN} \end{array}$$

```
var
  i: int;
  v: vector [100] of int;
  b: bool;

  b = i in v;
```

⇒

```
LOD 0 ^i
LOD 0 ^v
IN
STO 0 ^b
```

Arithmetic Operations: +, –, *, /

$math\text{-}expr \rightarrow expr_1 \ \ expr_2$

	+	–	*	/
int	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ `IPLUS`	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ `IMINUS`	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ `ITIMES`	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ `IDIV`
real	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ `RPLUS`	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ `RMINUS`	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ `RTIMES`	$\langle expr_1 \rangle$ $\langle expr_2 \rangle$ `RDIV`

```
var
  i, j, k: int;
  x, y: real;

  i = (i + 5) * (j – k);
  x = (y + 3.14) / (x + y);
```

```
LOD 0 ^i
LDI 5
IPLUS
LOD 0 ^j
LOD 0 ^k
IMINUS
ITIMES
STO 0 ^i
LOD 0 ^y
LDR 3.14
RPLUS
LOD 0 ^x
LOD 0 ^y
RPLUS
RDIV
STO 0 ^x
```

Negation Operations: **-**, **not**

neg-expr → *expr*

- (int)	**- (real)**	**NOT**
⟨ *expr* ⟩	⟨ *expr* ⟩	⟨ *expr* ⟩
IUMI	RUMI	NEG

```
var
  i, j, k: int;
  a, b: bool;

  b = i > j * k and not (a or j == -k);
```

```
LOD 0 ^i
LOD 0 ^j
LOD 0 ^k
ITIMES
IGT
JMF 11
LOD 0 ^a
JMF 3
LDC '1'
JMP 5
LOD 0 ^j
LOD 0 ^k
IUMI
EQU
NEG
JMP 2
LDC '0'
STO 0 ^b
```

Output Operation: `wr`

wr-expr $\rightarrow$ *specifier-opt expr*

⇨

null specifier

⟨ *expr* ⟩
`WR` *format*

specifier instantiated

⟨ *expr* ⟩
⟨ *specifier* ⟩
`FWR` *format*

```
var
  i, j, k: int;
  r: struct(a: int; b: real;);
  v: vector [10] of string;

  x = i – (wr (i + j));
  r2 = wr ["r.dat"] r;
  v2 = wr v;
```

⇨

```
LOD 0 ^i
LOD 0 ^i
LOD 0 ^j
IPLUS
WR "i"
IMINUS
STO 1 ^x
LOD 0 ^r
LDS "r.dat"
FWR "(a:i,b:r)"
STO 2 ^r2
LOD 0 ^v
WR "[10,s]"
STO 3 ^v2
```

EBNF of format

format $\rightarrow$ *atomic-format* | *struct-format* | *vector-format*
atomic-format $\rightarrow$ **c** | **i** | **r** | **s** | **b**
struct-format $\rightarrow$ **(***attr*{**,***attr*}**)**
attr $\rightarrow$ **id:***format*
vector-format $\rightarrow$ **[num,***format***]**

- **Notes:**
  - *specifier* = expression of file name
  - Argument of `WR`, `FWR`: string specifying operand schema

Function Call

func-call → **id** { $expr_i$ } ⇨

```
⟨ expr1 ⟩
⟨ expr2 ⟩
   …
⟨ exprn ⟩
PUSH  num-formals  num-locals  chain
  GOTO  entry
POP
```

Scenario	Chain
f1 parent of f2	0
f1 sibling of f2	1
otherwise	Level(f1) - Level(f2) + 1

```
var
  i, j, k: int;
  r: struct(a: int; b: string;);

  k = j - f(i+j, x, r.b);
```

⇨

```
LOD  0  ^j
LOD  0  ^i
LOD  0  ^j
IPLUS
LOD  1  ^x
LDA  0  ^r
FDA  |int|
EIL  |string|
PUSH  3  7  2
  GOTO  ^f
POP
IMINUS
STO  0  ^k
```

f1 calls f2

- **Notes:**
  - *num-formals* = number of formals parameters
  - *num-locals* = number of (non-parameter) local objects
  - *chain* = distance bewteen the caller environment and the environment where the function is defined
  - *entry* = address of entry point of function (body)

Dynamic Input: `rd`

rd-expr → *specifier-opt domain*

⇨

`RD` *format*	null specifier
⟨*specifier*⟩ `FRD` *format*	specifier instantiated

```
type
  Vect: vector [100] of real;
var
  i, j: int;
  v: Vect;

  j = i + rd int;
  v = f(rd [s] Vect);
```

⇨

```
LOD 0 ^i
RD "i"
IPLUS
STO 0 ^j
LOD 1 ^s
FRD "[100,r]"
PUSH 8 2
  GOTO ^f
POP
STO 0 ^v
```

Conditional Expression

$$cond\text{-}expr \rightarrow expr_1\ \ expr_2\ \ elsif\text{-}expr\text{-}list\text{-}opt\ \ expr_3$$
$$elsif\text{-}expr\text{-}list\text{-}opt \rightarrow \{\ expr_{1i},\ expr_{2i}\ \}$$

```
⟨expr1⟩
JMF  next1
⟨expr2⟩
JMP  exit1
⟨expr11⟩
JMF  next2
⟨expr21⟩
JMP  exit2
⟨expr12⟩
JMF  next3
⟨expr22⟩
JMP  exit3
   …
⟨expr1n⟩
JMF  next_n+1
⟨expr2n⟩
JMP  exit_n+1
⟨expr3⟩
```

```
var
  i, j, k: int;
  a, b, c: bool;

  w = if a or b then
        i + 48;
      elsif c and j > 20 then
        j – k;
      elsif i != j / k then
        k * i;
      else
        36;
      endif;
```

⇨

```
LOD 0 ^a
JMF 3
LDC '1'
JMP 2
LOD 0 ^b
JMF 5
LOD 0 ^i
LDI 48
IPLUS
JMP 24
LOD 0 ^c
JMF 5
LOD 0 ^j
LDI 20
IGT
JMP 2
LDC '0'
JMF 5
LOD 0 ^j
LOD 0 ^k
IMINUS
JMP 12
LOD 0 ^i
LOD 0 ^j
LOD 0 ^k
IDIV
NEQ
JMF 5
LOD 0 ^k
LOD 0 ^i
ITIMES
JMP 2
LDI 36
```

- **Notes:**
  - $next_i$ = distance from next condition
  - $exit_i$ = distance from exit of conditional expression

Built-in Functions: **`TOINT`**, **`TOREAL`**

built-in-call → *expr* ⇒

```
⟨ expr⟩
TOINT
```

```
⟨ expr⟩
TOREAL
```

```
var
  i, j: int;
  x: real;

  x = toreal(i+j);
```

⇒

```
LOD 0 ^i
LOD 0 ^j
IPLUS
TOREAL
STO 0 ^x
```

Input Statement: `read`

read-stat → *specifier-opt* **id** ⇨

`READ` *offset-env oid format*	null specifier
⟨*specifier*⟩ `FREAD` *offset-env oid format*	specifier instantiated

```
type
  Vect: vector [10] of real;

var
  k: int;
  s: string;
  r: struct(b: real; c: bool;);
  x: real;

  read k;
  read r;
  read [ s ] x;
  read v;
```

⇨

```
READ 0 ^k "i"
READ 0 ^r "(b:r,c:b)"
LOD 0 ^s
FREAD 0 ^x "r"
READ 2 ^v "[10,r]"
```

- **Notes:**
  - *oid* = identifier of object to instantiate
  - *offset-env* = distance in static chain
  - *format* = string specifying schema of object to instantiate

Output Statement: `write`

write-stat → *specifier-opt expr* ⇨

⟨*expr*⟩
`WRITE` *format*

null specifier

⟨ *expr*⟩
⟨ *specifier*⟩
`FWRITE` *format*

specifier instantiated

```
var
  i: int;
  s: string;
  r: struct(a: int; b: string;);

  write i + 25;
  write [ s ] r;
```

⇨

```
LOD 0 ^i
LDI 25
IPLUS
WRITE "i"
LOD 0 ^r
LOD 0 ^s
FWRITE "(a:i,b:s)"
```

- **Notes:**
  - *expr* = object to print
  - *specifier* (if instantiated) = name of file on which to print the instance
  - *format* = string specifying schema of object to instantiate

Conditional Statement: `if`

if-stat $\rightarrow$ *expr* *stat-list*$_1$ *elsif-stat-list-opt* [*stat-list*$_2$]
elsif-stat-list-opt $\rightarrow$ { *expr*$_{1i}$, *stat-list*$_{2i}$ }

```
⟨expr⟩
JMF  next1
⟨stat-list1⟩
JMP  exit1

⟨expr11⟩
JMF  next2
⟨stat-list21⟩
JMP  exit2

⟨expr12⟩
JMF  next3
⟨stat-list22⟩
JMP  exit3

  ...

⟨expr1n⟩
JMF  next(n+1)
⟨stat-list2n⟩
JMP  exit(n+1)

[⟨stat-list2⟩]
```

```
var
  i, j, k: int;

  if i == j then
    j = j + 3;
  elsif i > j then
    i = i - 2;
    j = i + k ;
  else
    i = j * k;
  endif;
```

$\Rightarrow$

```
LOD 0 ^i
LOD 0 ^j
EQU
JMF 6
LOD 0 ^j
LDI 3
IPLUS
STO 0 ^j
JMP 18
LOD 0 ^i
LOD 0 ^j
IGT
JMF 10
LOD 0 ^i
LDI 2
IMINUS
STO 0 ^j
LOD 0 ^i
LOD 0 ^k
IPLUS
STO 0 ^j
JMP 5
LOD 0 ^j
LOD 0 ^k
ITIMES
STO 0 ^i
```

- **Note:**
  - If *stat-list*$_2$ not specified $\rightarrow$

```
  ...
⟨expr1n⟩
JMF  next(n+1)
⟨stat-list2n⟩
```

`While` Loop

while-stat $\rightarrow$ *expr* *stat-list*

$\Rightarrow$

```
⟨ expr ⟩
JMF exit
⟨ stat-list⟩
JMP up
```

```
a, b, ris: int;
…
ris = 0;
while a >= b do
    ris = ris + 1;
    a = a – b;
endwhile;
```

$\Rightarrow$

```
LDI 0
STO 0 ^ris
LOD 0 ^a
LOD 0 ^b
IGE
JMF 10
LOD 0 ^ris
LDI 1
IPLUS
STO 0 ^ris
LOD 0 ^a
LOD 0 ^b
IMINUS
STO 0 ^a
JMP -12
```

For Loop

for-stat → **id** $expr_1$ $expr_2$ *stat-list*

⇨

id = $expr_1$;
temp = $expr_2$;
while **id** <= temp **do**
  stat-list;
  id = **id** +1;
endwhile;

⇨

⟨$expr_1$⟩
STO *env-offset* ^**id**
⟨$expr_2$⟩
STO 0 ^temp
LOD *env-offset* ^**id**
LOD 0 ^temp
ILE
JMF *down*
⟨*stat-list*⟩
LOD *env-offset* ^**id**
LDI 1
IPLUS
STO *env-offset* ^**id**
JMP *up*

```
for i=j+2 to j*k do
    j = j-k;
endfor;
```

⇨

```
LOD 0 ^j
LDI 2
IPLUS
STO 0 ^i
LOD 0 ^j
LOD 2 ^k
ITIMES
STO 0 ^temp
LOD 0 ^i
LOD 0 ^temp
ILE
JMF 10
LOD 0 ^j
LOD 2 ^k
IMINUS
STO 0 ^j
LOD 0 ^i
LDI 1
IPLUS
STO 0 ^i
JMP -12
```

Foreach Loop

foreach-stat → **id** *expr* *stat-list*

⇒

```
% let n be the size of vector expr
% let i be an auxiliary local integer
% let temp be an auxiliary local var
i = 0; temp = expr;
repeat
  id = temp[i];
  stat-list;
  i = i+1
until i == n;
```

⇒

```
LDI 0
STO 0 ^i
⟨expr⟩
STO 0 ^temp
LDA 0 ^temp
LOD 0 ^i
IXA size
(E|S)IL size
STO env-offset ^id
⟨stat-list⟩
LOD 0 ^i
LDI 1
IPLUS
STO 0 ^i
LOD 0 ^i
LDI n
EQU
JMF up
```

```
v: vector [10] of int;
k: int;

foreach k in v do
    sum = sum + k;
endforeach;
```

⇒

```
LDI 0
STO 0 ^i
LOD 0 ^v
STO 0 ^temp
LOD 0 ^temp
LOD 0 ^i
IXA |int|
EIL |int|
STO 0 ^k
LOD 2 ^sum
LOD 0 ^k
IPLUS
STO 2 ^sum
LOD 0 ^i
LDI 1
IPLUS
STO 0 ^i
LOD 0 ^i
LDI n
EQU
JMF -16
```

Function Definition

function-decl → **id** *formal-list-opt domain*
type-sect-opt
var-sect-opt
const-sect-opt
func-list-opt
func-body

⇨

```
FUNC fid
⟨new-variables⟩
⟨new-constants⟩
⟨assign-constants⟩
⟨func-body⟩
```

```
func alpha(i,j,k: int;): int
  var
    n, m: int;
  const
    msg: string = "Hello!"
begin alpha
  write msg;
  n = i + k;
  m = n - k;
  j = n * m;
  k = k / j;
  return n+m-k;
end alpha;
```

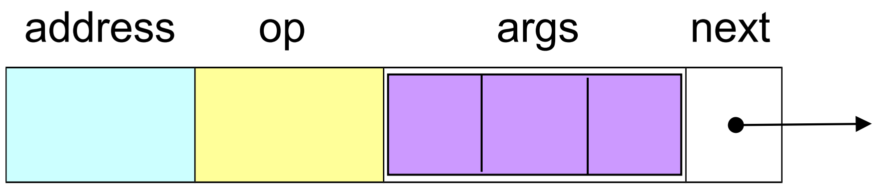
⇨

```
FUNC ^alpha
NEW |n|
NEW |m|
NEW |msg|
LDS "Hello!"
STO 0 ^msg
LOD 0 ^msg
WRITE "s"
LOD 0 ^i
LOD 0 ^k
IPLUS
STO 0 ^n
LOD 0 ^n
LOD 0 ^k
IMINUS
STO 0 ^m
LOD 0 ^n
LOD 0 ^m
ITIMES
STO 0 ^j
LOD 0 ^k
LOD 0 ^j
IDIV
STO 0 ^k
LOD 0 ^n
LOD 0 ^m
IPLUS
LOD 0 ^k
IMINUS
RETURN
```

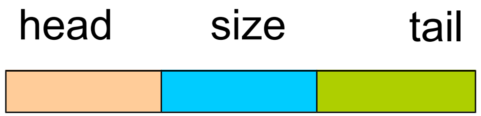
- **Notes:**
  - *fid* = function identifier
  - If **`return`** = non final instruction → `JMP` to `RETURN`

Data Structures for Code Generation

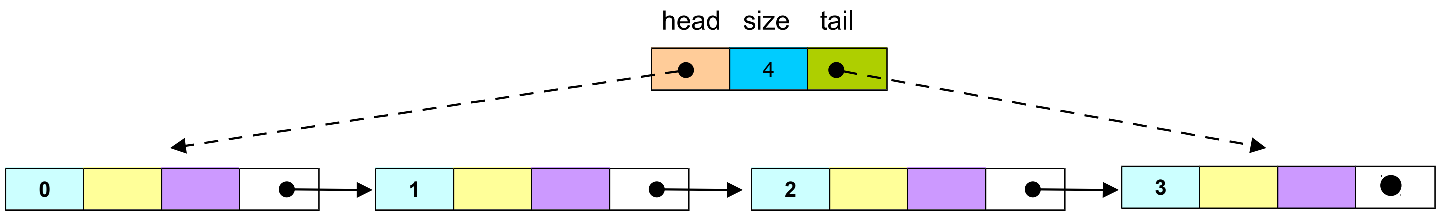
- `Stat`:



- `Code`:



- Representation of a segment of code (sequence of S-code statements):



gen.c

```
void relocate_address(Code code, int offset)
Code appcode(Code code1, Code code2)
Code endcode()
Code concode(Code code1, Code code2, ...)
Stat *newstat(Operator op)
Code makecode(Operator op)
Code makecode1(Operator op, int arg)
Code makecode2(Operator op, int arg1, int arg2)
Code make_push_pop(int nforms, int nlocs, int chain, int entry)
Code make_ldc(char c)
Code make_ldi(int i)
Code make_ldr(float r)
Code make_lds(char *s)
```

gen.c: `relocate_address()`

```
void relocate_address(Code code, int offset)
{
  Stat *pt = code.head;
  int i;

  for(i = 1; i <= code.size; i++)
  {
    pt->address += offset;
    pt = pt->next;
  }
}
```

gen.c: appcode()

```
Code appcode(Code code1, Code code2)
{
  Code rescode;

   relocate_address(code2, code1.size);
   rescode.head = code1.head;
   rescode.tail = code2.tail;
   code1.tail->next = code2.head;
   rescode.size = code1.size + code2.size;
   return rescode;
}
```

gen.c: endcode(), concode()

```c
Code endcode()
{
  static Code code = {NULL, 0, NULL};

  return code;
}

Code concode(Code code1, Code code2, ...)
{
  Code rescode = code1, *pcode = &code2;

  while(pcode->head != NULL)
  {
    rescode = appcode(rescode, *pcode);
    pcode++;
  }
  return rescode;
}
```

gen.c: newstat(), makecode(), makecode1()

```
Stat *newstat(Operator op)
{
  Stat *pstat;

  pstat = (Stat*)newmem(sizeof(Stat));
  pstat->address = 0;
  pstat->op = op;
  pstat->next = NULL;
  return pstat;
}
```

```
Code makecode(Operator op)
{
  Code code;

  code.head = code.tail = newstat(op);
  code.size = 1;
  return code;
}
```

```
Code makecode1(Operator op, int arg)
{
  Code code;

  code = makecode(op);
  code.head->args[0].ival = arg;
  return code;
}
```

gen.c: make_push_pop()

```
Code make_push_pop(int nforms, int nlocs, int chain, int entry)
{
    return concode(makecode2(S_PUSH, nforms nlocs, chain),
                   makecode1(S_GOTO, entry),
                   makecode(S_POP),
                   endcode());
}
```

gen.c: `make_lds()`

```
Code make_lds(char *s)
{
    Code code;

    code = makecode(S_LDS);
    code.head->args[0].sval = s;
    return code;
}
```

Architecture of S-machine