Compilers

3rd year
Spring term

Alfonso Ortega: alfonso.ortega@uam.es
Enrique Alfonseca: enrique.alfonseca@uam.es

Lesson 6 part a: generation of intermediate code

Intermediate Representations

INDEX

- **Introduction**
- **Postfix (suffix) notation**
- **Tuples**

Intermediate Representations

General concepts

- An **intermediate representation** describes the program,
  - that has been analysed in the precious steps, and represented implicitly or explicitly as an annotated parse tree,
  - using an abstract format which will be more similar to the format of the generated code.
  - We are going to study two different representations:
    - Suffix notation
    - Tuples

Intermediate Representations

Introductory examples

- Let us consider some introductory examples to intermediate representations, focusing on arithmetic expressions.
- Consider the following ASPLE expression:

  `a * ( 9 + d )`

- The following would be the equivalent postfix notation.

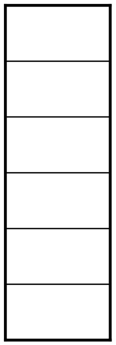
  `a9d+*`

- This notation receives several names: *postfix*, *suffix*, and *inverse polish notation*.
  - Any expression can be written unambiguously without parentheses.
  - We can build very easily interpreters for these expressions, using a stack.

Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

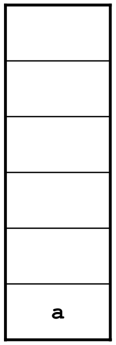
a * (9 + d) ⇔ a9d+*



Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

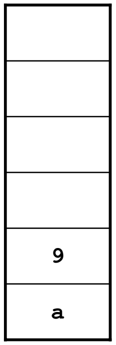
a * (9 + d) ⇔ a9d+*



Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

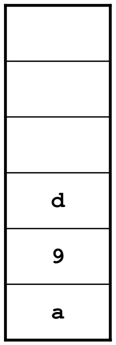
a * (9 + d) ⇔ a9d+*



Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

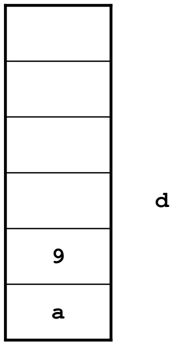
`a * ( 9 + d ) ⇔ a9d+*`



Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

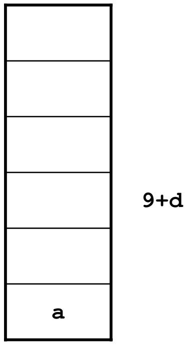
`a * ( 9 + d ) ⇔ a9d+*`



Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

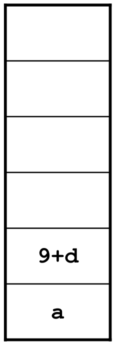
`a * ( 9 + d ) ⇔ a9d+*`



Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

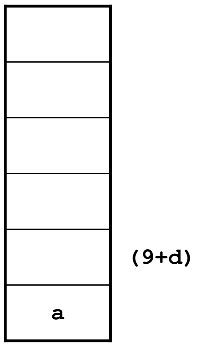
`a * ( 9 + d ) ⇔ a9d+*`



Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

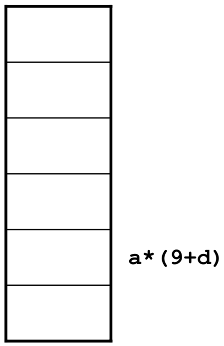
a * (9 + d) ⇔ a9d+*



Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

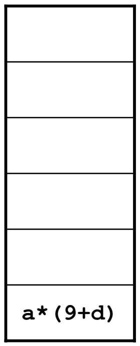
a * (9 + d) ⇔ a9d+*



Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

a * (9 + d) ⇔ a9d+*



Intermediate Representations

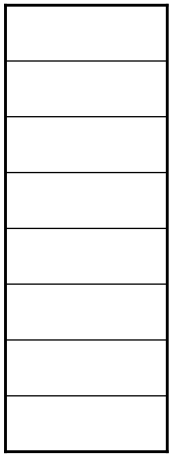
Introductory examples: conclusions of the first example

- We can draw the following conclusions from *postfix* notation:
  - The notation can simplify some features of the languages: ambiguity, parentheses, etc.
  - It simplifies the evaluation, using a stack.
- We could wonder whether this notation can be extended to all the other constructions present in high-level languages.
  - For instance, could it be used as well for assignments?
- This format, if feasible, could be a suitable intermediate representation:
  - It is simple
  - It is easy to interpret
  - It is unambiguous

Intermediate Representations

Introductory examples: extension to other types of expressions

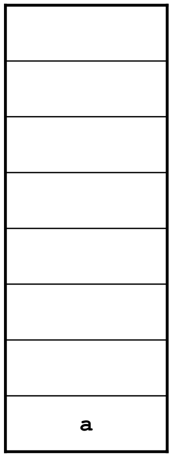
`a := a * ( 9 + d )` ⇔ `aa9d+*:=`



Intermediate Representations

Introductory examples: extension to other types of expressions

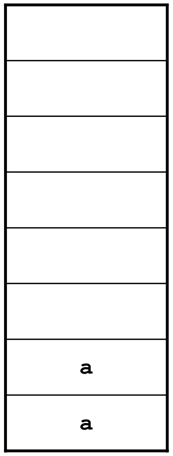
`a := a * ( 9 + d )` ⇔ `aa9d+*:=`



Intermediate Representations

Introductory examples: extension to other types of expressions

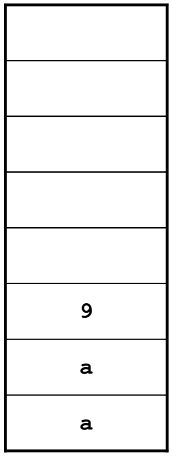
`a := a * ( 9 + d )` ⇔ `aa9d+*:=`



Intermediate Representations

Introductory examples: extension to other types of expressions

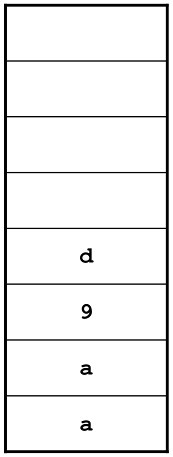
`a := a * ( 9 + d )` ⇔ `aa9d+*:=`



Intermediate Representations

Introductory examples: extension to other types of expressions

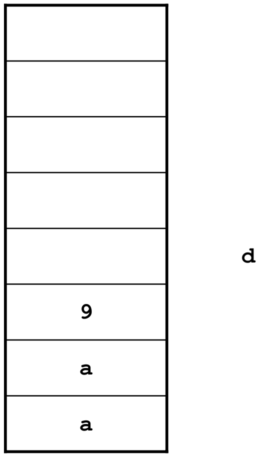
a := a * (9 + d) ⇔ aa9d+*:=



Intermediate Representations

Introductory examples: extension to other types of expressions

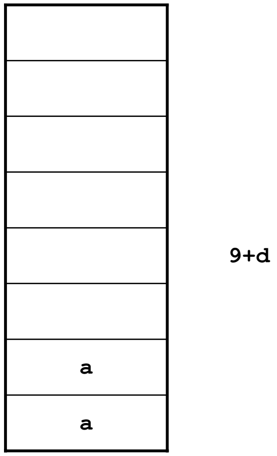
a := a * (9 + d) ⇔ aa9d+*:=



Intermediate Representations

Introductory examples: extension to other types of expressions

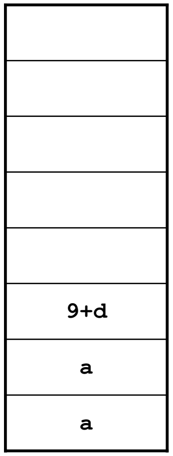
a := a * (9 + d) ⇔ aa9d+*:=



Intermediate Representations

Introductory examples: extension to other types of expressions

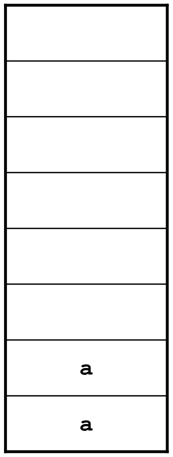
a := a * (9 + d) ⇔ aa9d+*:=



Intermediate Representations

Introductory examples: extension to other types of expressions

a := a * (9 + d) ⇔ aa9d+*:=



a

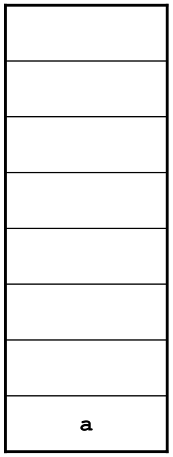
a

(9+d)

Intermediate Representations

Introductory examples: extension to other types of expressions

a := a * (9 + d) ⇔ aa9d+*:=



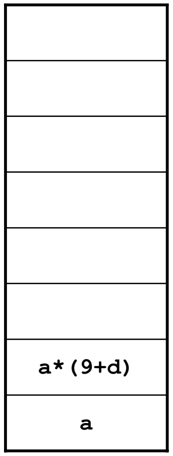
a

a*(9+d)

Intermediate Representations

Introductory examples: extension to other types of expressions

a := a * (9 + d) ⇔ aa9d+*:=



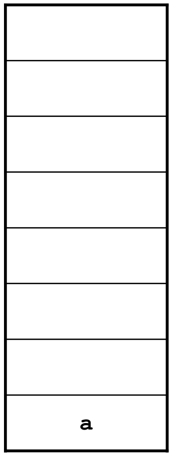
a*(9+d)

a

Intermediate Representations

Introductory examples: extension to other types of expressions

a := a * (9 + d) ⇔ aa9d+*:=



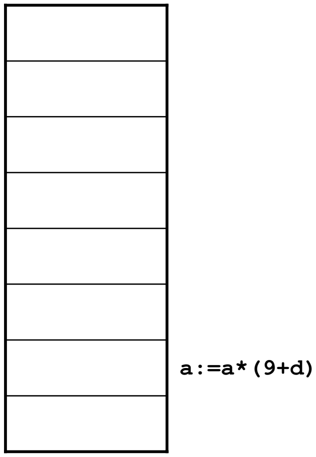
a

a*(9+d)

Intermediate Representations

Introductory examples: extension to other types of expressions

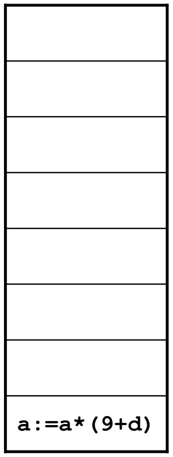
`a := a * ( 9 + d )` ⇔ `aa9d+*:=`



Intermediate Representations

Introductory examples: extension to other types of expressions

`a := a * ( 9 + d )` ⇔ `aa9d+*:=`



Intermediate Representations

Introductory examples: TUPLES

- The other possibility that we are going to study, apart from the suffix notation, are the tuples:
  - We can “reduce” all operators to binary and unary operators.
  - The intermediate representation can be considered as a sequence of partial executions, where the result of each step is stored in a temporal variable.
    - Similar to the way in which a child has to perform the operations one by one, annotating the intermediate result to use it afterwards.
  - We could annotate, for each intermediate operation:
    - The operation.
    - The operand, or the two operands.
    - The place where the result will be stored.

- This example...

```
a := a * ( 9 + d )
```

- could be represented with the following sequence of steps (quadruples):

```
(+ ,9 ,d ,output_step1 )
(* ,a ,output_step2 ,a )
```

Intermediate Representations

General concepts

- When we use an intermediate representation before the code generation step, the following are two common representations:
  - Suffix notation
    - The expressions are transformed into suffix notation.
    - The technique is extended to the other constructions in the language:
      - Assignments
      - Control-flow structures.
    - The only requirement to implement them is to use a stack.
    - This is the procedure followed by most assemblers.
  - Tuples (quadruples and triples)
    - We start from a standard notation for binary arithmetic expressions using tuples.
    - The notation is next extended to the other constructions in the language.

Intermediate Representations

INDEX

- **Introduction**
- **Postfix (suffix) notation**
- **Tuples**

Suffix notation

General concepts: abstract stack machines

- Suffix notation is really an intermediate representation of the program, which can be considered as a compiled program for an ***abstract stack machine***.
- This is a machine with the following **storage modules:**
  - Independent memories:
    - For the executable instructions.
    - For the data used by them.
  - A stack
    - To perform all the arithmetic and logic operations.
- In the following slides, we describe how this machine works.

Suffix notation

General concepts: abstract stack machines

- It is usually assumed that the machine has the following **operations**:
  - Each logic or arithmetic operation will be assumed to be directly supported by the machine:

    `+`, `*`, `/`, `-`, `and`, `or`

    - as shown before in the first example of suffix notation.
  - To be able to execute assignment operations, it has to treat **identifiers**:
    - It is different to have an identifier at the **left-hand-side** of an assignment (in this case it represents a memory zone to store a value). We can call this operation:

      `left_value identifier`

    - If an identifier appears at the **right-hand-side** of the assignment, it represents the value that will be assigned. We can call it:

      `right_value identifier`

Suffix notation

General concepts: abstract stack machines

- Operations for the **stack management**:
  - To insert a value into the stack: `push value`
  - To eliminate the value from the stack: `pop`
- Other **statements**:
  - Assignments, as binary operators:

    `identifier := expression`

  - Control-flow operations:
    - Definitions of labels for jumps:

      `define_label label` or `label:`(*)

    - Unconditional jumps:

      `jump_to label`

    - Conditional jumps:

      `jump_if_false label`

      `jump_if_zero label`

Suffix notation

General concepts: abstract stack machines: IMPLEMENTATION

- If you have experience with assembly languages:
  - The operations in the *abstract stack machine* are very similar to the operations available by most assemblers.
  - The data structure (a stack) also appears in assembly languages.
- Therefore, this intermediate representation, based on suffix notation, will allow an easy translation into any assembly language for any platform.

Suffix notation

General concepts: abstract stack machines

- Any compiler using suffix notation as intermediate representation has to include two different steps:
  - The generation (for instance, in an independent file) of the suffix representation equivalent to the analysed program.
  - A translation of this suffix representation into the output language of the compiler: the generation of the final code.
- Right now, we shall focus on some issues about the code generation for both steps.
- We shall only suggest some of the translation issues.

Suffix notation

Generation of the suffix notation associated to a program

- As indicated, the generation of the intermediate representation takes place where the designer of the compiler decides, during the syntactic or the semantic analysis.
- The conversion can be done in the semantic actions associated to the rules.
- The technique will be explained with the following example (arithmetic expressions). Consider the following grammar:

```
E→E+T
E→E-T
E→T
T→T*F
T→T/F
T→F
F→i
F→(E)
F→-F
```

Suffix notation

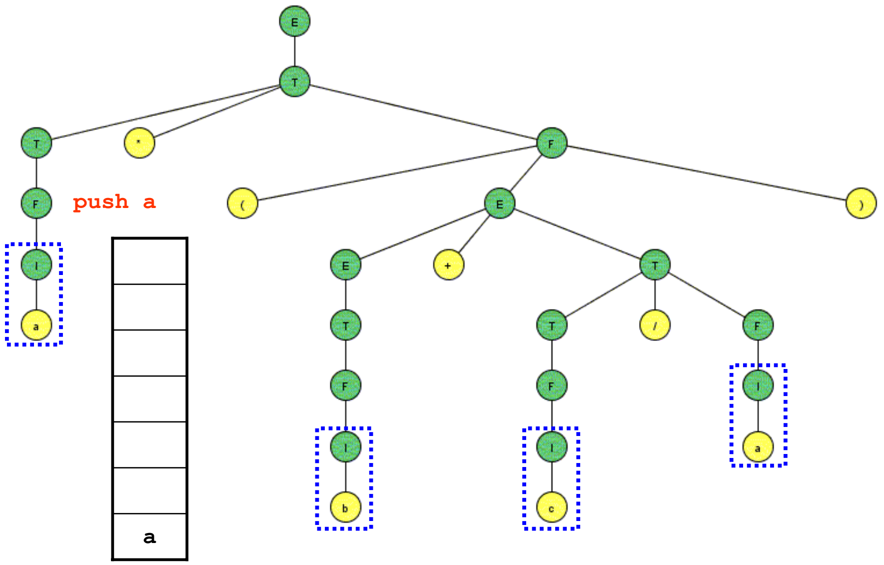
Generation of the suffix notation associated to a program

- We shall extend the grammar with the following semantic actions in the abstract stack machine:

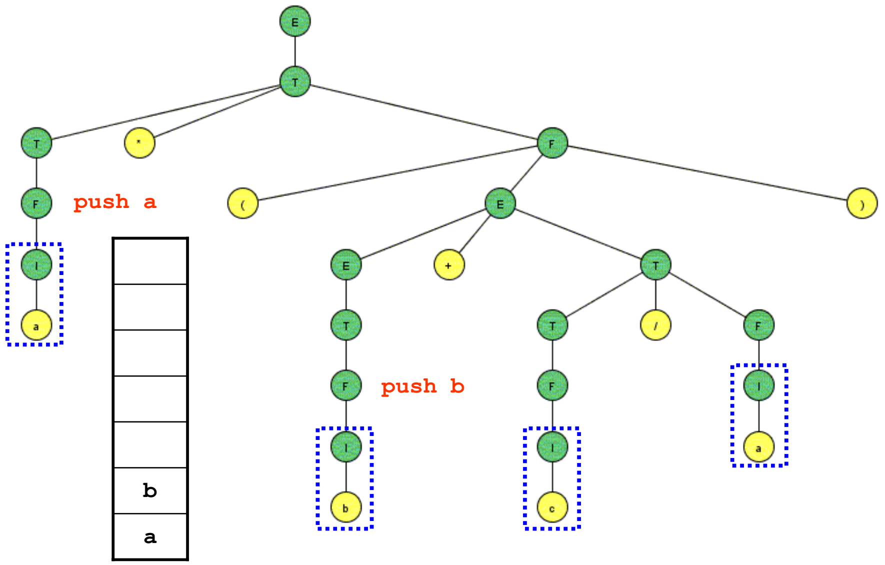
```
E→E+T {push +}
E→E-T {push -}
E→T
T→T*F {push *}
T→T/F {push /}
T→F
F→i   {push i}
F→(E)
F→-F  {push _}(*)
```

(*) Although the same symbol represents the subtraction and the negation, they are different operations, so we use different symbols for the abstract stack machine.

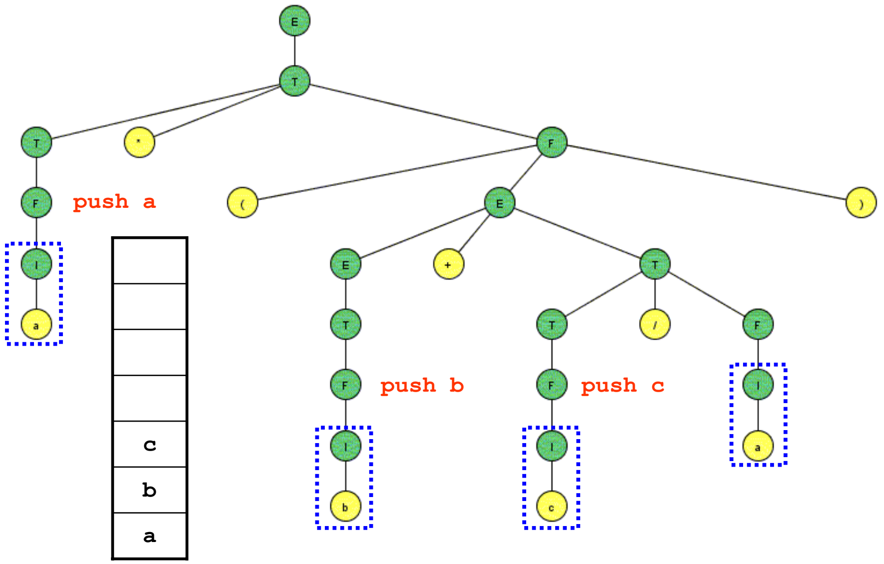
Suffix notation



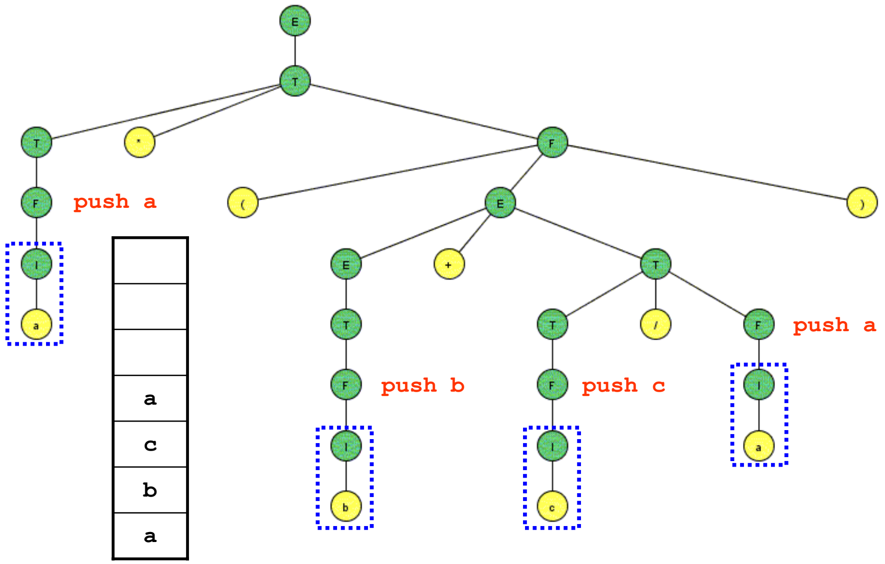
Suffix notation



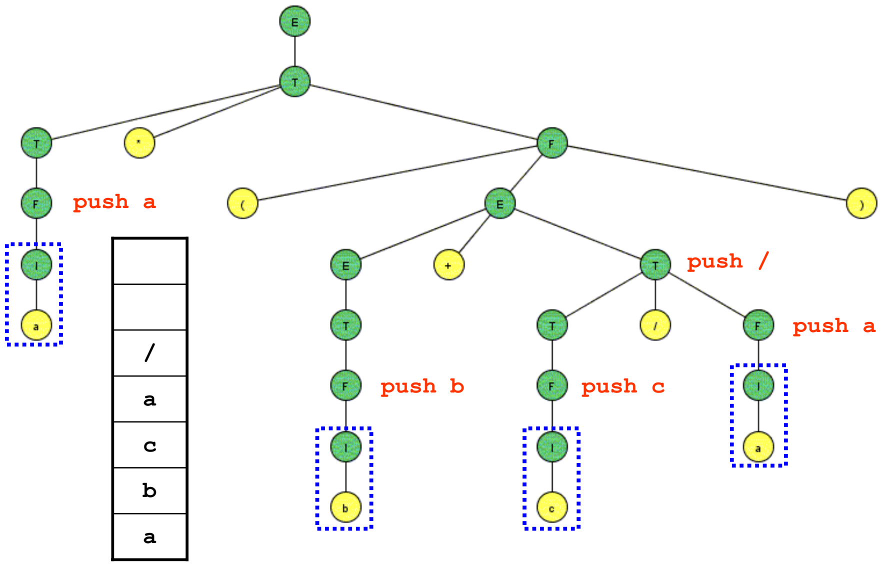
Suffix notation



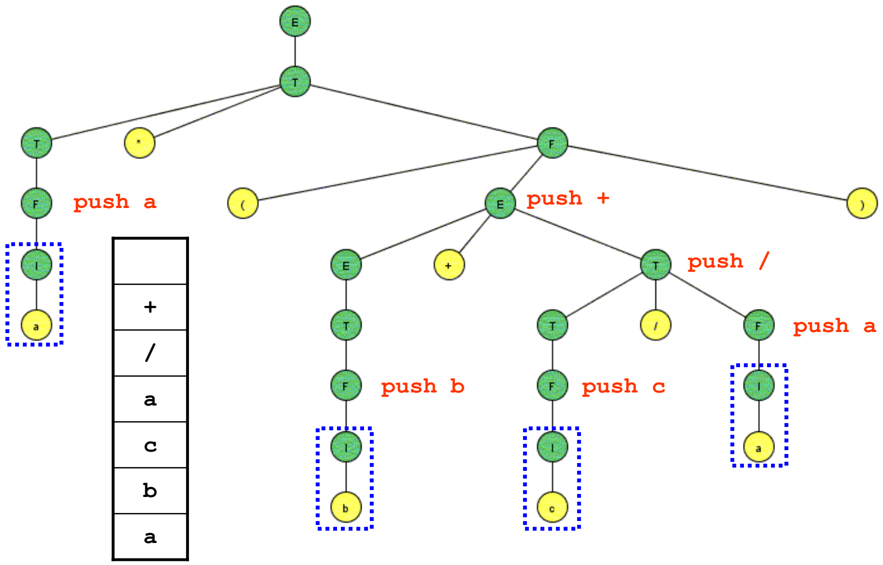
Suffix notation



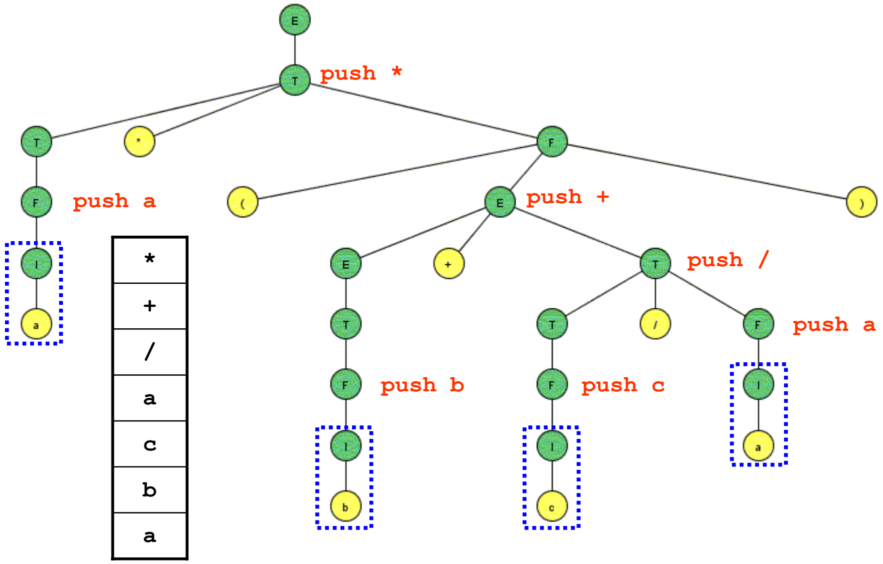
Suffix notation



Suffix notation



Suffix notation



Suffix notation

Generation of the suffix notation associated to a program

- Therefore, if we treat the stack, at this point as the intermediate representation file, it will contain the following information:

a	b	c	a	/	+	*	

Suffix notation

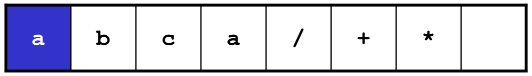
Code generation, associated to the intermediate representation

- Informally, we have shown the way the abstract stack machine works:
  - For any **operand**:
    - It is pushed into the stack
  - For any **operator**:
    - Pop from the stack as many operands as arguments the operator requires.
    - Perform the operation.
    - Push the result.

Suffix notation

Code generation, associated to the intermediate representation

- In our example:



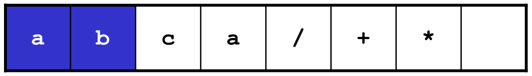
a	b	c	a	/	+	*	

```
push dword [a]
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example:



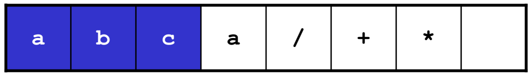
a	b	c	a	/	+	*	

```
push dword [a]
push dword [b]
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example:



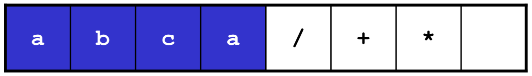
a	b	c	a	/	+	*	

```
push dword [a]
push dword [b]
push dword [c]
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example:



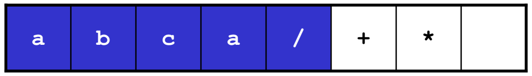
a	b	c	a	/	+	*	

```
push dword [a]
push dword [b]
push dword [c]
push dword [a]
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example:



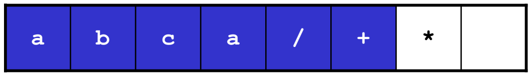
a	b	c	a	/	+	*	

```
push dword [a]
push dword [b]
push dword [c]
push dword [a]
pop  eax
pop  ebx
idiv ebx, eax
push eax
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example:



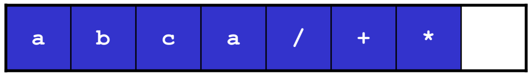
a	b	c	a	/	+	*	

```
push dword [a]
push dword [b]
push dword [c]
push dword [a]
pop  eax
pop  ebx
idiv ebx, eax
push eax
pop  eax
pop  ebx
add  ebx, eax
push eax
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example:



a	b	c	a	/	+	*	

```
push dword [a]
push dword [b]
push dword [c]
push dword [a]
pop  eax
pop  ebx
idiv ebx, eax
push eax
pop  eax
pop  ebx
add  ebx, eax
push eax
pop  eax
pop  ebx
mul  ebx, eax
push eax
```

Suffix notation

Code generation, associated to the intermediate representation

- The following “attributes grammar” generates the sequence of PC Assembly instructions:

```
<Operand>→i {printf("push dword %s\n",i.name) (1)}
<Operand>→cte {printf("push dword %s\n",cte.name) (1)}
<Operand>→<Operand> <Operand> <Dyadic Operator>
       {if ("the operation is valid, (types, etc...)")
        {
         printf("pop eax\n"),
         printf("pop ebx\n"),
         printf("%s ebx eax\n",<Dyadic Operator>.symbol) (2)
         printf("push eax\n")}
        else "ERROR"
       }
<Operand>→<Operand> <Monadic Operator>
       {if ("the operation is valid, (types, etc...)")
        {printf("pop eax\n"),
         printf("%s eax\n",<Monadic Operator>.symbol)
         printf("push eax\n")}}
```

Suffix notation

Code generation, associated to the intermediate representation

- Formal representation:
  - (1) We assume that **`cte`** and **`i`** has their semantic value in the attribute “name”
  - (2) We assume that the “tokens” **`<Dyadic operator>`** and **`<Monadic operator>`** have the symbol of the operation in assembly language as their semantic value.
  - This new grammar, which is very simple, can be applied to the generated representation in post-fix notation.
    - It would be the grammar for a “second compiler” that translates the suffix notation into assembly language.
  - It will generate the equivalent Assembly instructions.

Suffix notation

Code generation, associated to the intermediate representation

- This idea can be extended to all the other kinds of statements..
- The following are some brief descriptions of the application of this technique to compile assignments, and to conditional and unconditional jumps. The other constructions (if-then-else, loops, etc.) can be translated in a following way.

Code generation, associated to the intermediate representation: assignment

- Concerning assignment, as we said, it could be translated into suffix notation considering it as another operator.

`identifier := expression`

- Consider this statement for assigning a value to **`a`**.

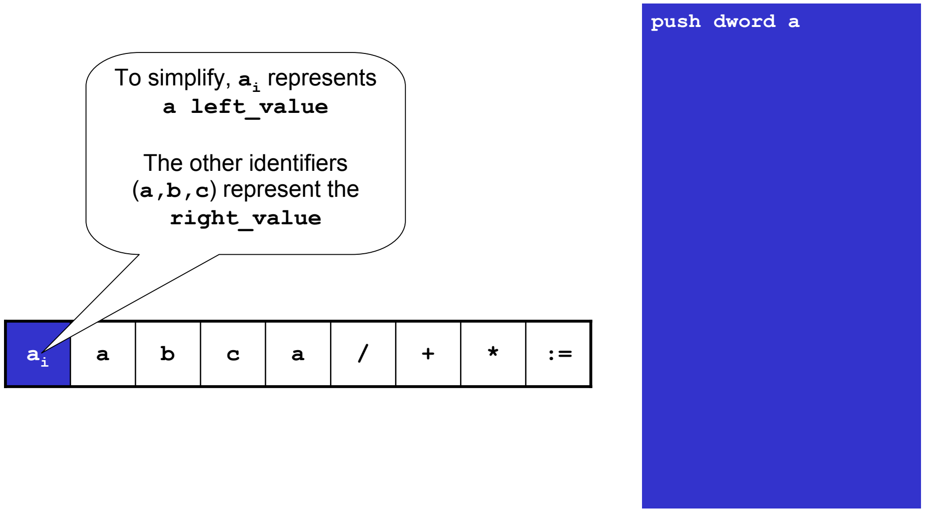
`a := a*(b+c/a)`

`aabca/+*:=`

- The actions that generate the final code have to ensure that they only occur if there is no error (type compatibility, etc.)

Suffix notation

Code generation, associated to the intermediate representation



To simplify, **a**$_i$ represents **`a left_value`**

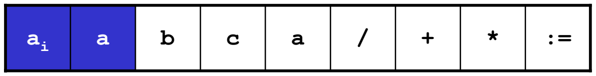
The other identifiers (**`a,b,c`**) represent the **`right_value`**

a_i	a	b	c	a	/	+	*	:=

```
push dword a
```

Suffix notation

Code generation, associated to the intermediate representation

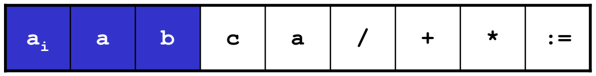


a_i	a	b	c	a	/	+	*	:=

```
push dword a
push dword [a]
```

Suffix notation

Code generation, associated to the intermediate representation

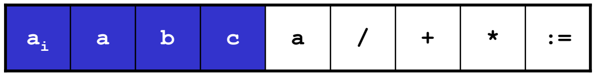


a_i	a	b	c	a	/	+	*	:=

```
push dword a
push dword [a]
push dword [b]
```

Suffix notation

Code generation, associated to the intermediate representation

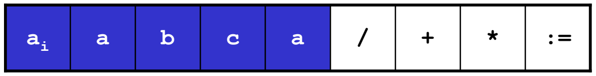


a_i	a	b	c	a	/	+	*	:=

```
push dword a
push dword [a]
push dword [b]
push dword [c]
```

Suffix notation

Code generation, associated to the intermediate representation



a_i	a	b	c	a	/	+	*	:=

```
push dword a
push dword [a]
push dword [b]
push dword [c]
push dword [a]
```

Suffix notation

Code generation, associated to the intermediate representation

a_i	a	b	c	a	/	+	*	:=

```
push dword a
push dword [a]
push dword [b]
push dword [c]
push dword [a]
pop  eax
pop  ebx
idiv ebx, eax
push eax
```

Suffix notation

Code generation, associated to the intermediate representation

a_i	a	b	c	a	/	+	*	:=

```
push dword a
push dword [a]
push dword [b]
push dword [c]
push dword [a]
pop  eax
pop  ebx
idiv ebx, eax
push eax
pop  eax
pop  ebx
add  ebx, eax
push eax
```

Suffix notation

Code generation, associated to the intermediate representation

a_i	a	b	c	a	/	+	*	:=

```
push dword a
push dword [a]
push dword [b]
push dword [c]
push dword [a]
pop  eax
pop  ebx
idiv ebx, eax
push eax
pop  eax
pop  ebx
add  ebx, eax
push eax
pop  eax
pop  ebx
mul  ebx
push eax
```

Suffix notation

Code generation, associated to the intermediate representation

a_i	a	b	c	a	/	+	*	:=

```
push dword a
push dword [a]
push dword [b]
push dword [c]
push dword [a]
pop  eax
pop  ebx
idiv ebx, eax
push eax
pop  eax
pop  ebx
add  ebx, eax
push eax
pop  eax
pop  ebx
mul  ebx
push eax
pop  ebx
pop  eax
mov  dword [eax], ebx
```

Suffix notation

Code generation, associated to the intermediate representation: inconditional jump

- Remember the syntax of the unconditional jumps in the abstract stack machine:

```
jump_to <Label>
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example,

Label	jump_to

```
push dword Label
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example,

Label	jump_to

```
push dword Label
pop eax
jmp near eax
```

Suffix notation

Code generation, associated to the intermediate representation: conditional jump

- Remember the syntax of the conditional jumps:

```
"condition"
jump_if_false <Label>
```

- It is frequent to consider separately the two elements of the conditional jump:
  - The evaluation of the condition. In most low-level languages, we can assume that the evaluation leaves some state flags which reflect the result of this evaluation.
  - The jump to the label.
- The condition can be evaluated in a similar way as the arithmetic expressions that we have already seen.
- The conditional statement will be processed as follows:

Suffix notation

Code generation, associated to the intermediate representation

- In our example,

Label	jump_if_false

```
"code for the condition,
 which modifies the flags"
push dword Label
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example

Label	jump_if_false

```
"code for the condition,
 which modifies the flags"
push dword Label
pop eax
jz near eax(*)
```

Suffix notation

Code generation, associated to the intermediate representation: if-them

- The following example shows how we could treat a slightly more complex statement.
- Imagine a high-level language statement, e.g.:

`if <p> then <inst1> else <inst2>`

- It could be transformed into suffix notation in the following way:

`<p> L1 jump_if_false <inst1> L2 jump_to L1: <inst2> L2:`

- The following example shows how this could be treated:

Suffix notation

Code generation, associated to the intermediate representation

- In our example

p	L1	jump_if_false	inst1	L2	jump_to	L1:	inst2	L2:

```
"code for p, which
 modifies the flags"
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example

p	L1	jump_if_false	inst1	L2	jump_to	L1:	inst2	L2:

```
"code for p, which
 modifies the flags"
push dword L1
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example

p	L1	jump_if_false	inst1	L2	jump_to	L1:	inst2	L2:

```
"code for p, which
 modifies the flags"
push dword L1
pop   eax
jz    near eax (*)
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example

p	L1	jump_if_false	inst1	L2	jump_to	L1:	inst2	L2:

```
"code for p, which
 modifies the flags"
push dword L1
pop   eax
jz    near eax (*)
"code for inst1"
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example

p	L1	jump_if_false	inst1	L2	jump_to	L1:	inst2	L2:

```
"code for p, which
 modifies the flags"
push dword L1
pop   eax
jz    near eax (*)
"code for inst1"
push dword L2
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example

p	L1	jump_if_false	inst1	L2	jump_to	L1:	inst2	L2:

```
“code for p, which
 modifies the flags”
push dword L1
pop  eax
jz   near eax (*)
“code for inst1”
push dword L2
pop  eax
jmp  near eax
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example

p	L1	jump_if_false	inst1	L2	jump_to	L1:	inst2	L2:

```
“code for p, which
 modifies the flags”
push dword L1
pop  eax
jz   near eax (*)
“code for inst1”
push dword L2
pop  eax
jmp  near eax
L1:
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example

p	L1	jump_if_false	inst1	L2	jump_to	L1:	inst2	L2:

```
“code for p, which
 modifies the flags”
push dword L1
pop  eax
jz   near eax (*)
“code for inst1”
push dword L2
pop  eax
jmp  near eax
L1:
“code for inst2”
```

Suffix notation

Code generation, associated to the intermediate representation

- In our example

p	L1	jump_if_false	inst1	L2	jump_to	L1:	inst2	L2:

```
“code for p, which
 modifies the flags”
push dword L1
pop  eax
jz   near eax (*)
“code for inst1”
push dword L2
pop  eax
jmp  near eax
L1:
“code for inst2”
L2:
```

Intermediate Representations

INDEX

- **Introduction**
- **Postfix (suffix) notation**
- **Tuples**

Generation of tuples

Informal introduction

- As always, we shall start with an introductory example about generation of quadruples, using the two different analysis procedures studied in this course.

Generation of tuples in bottom-up analysis

Introduction

- As we indicated before, synthesised attributes are compatible with bottom-up analysis.
- It suffices, then, to specify the semantic actions of this type, which will generate the quadruples while we perform the bottom-up analysis.
- The following example illustrates this idea:

Generation of tuples in bottom-up analysis

Example

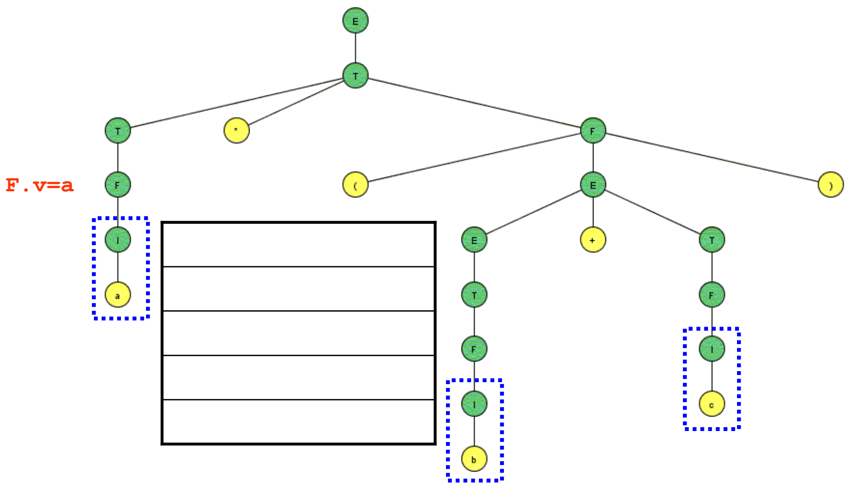
- Consider the following attributes grammar:

```
G3 =<{E.v,T.v,F.v},{i.name,+.símb,-.símb,*.símb,/.símb,(,)}
  {E→E+T {“If valid +” add(“+.symbol,E.v,T.v,id=new_symbol()”),
          E.v=id}
   E→E-T {“If valid -” add(“-.symbol,E.v,T.v,id=new_symbol()”),
          E.v=id}
   E→T {E.v=T.v}
   T→T*F {“If valid *” add(“*.symbol,T.v,F.v,id=new_symbol()”),
          T.v=id}
   T→T/F {“If valid /” add(“/.symbol,T.v,T.v,id=new_symbol()”),
          T.v=id}
   T→F {T.v=F.v}
   F→i {F.v=i.name}
   F→(E) {F.v=E.v}
   F→-F{“If valid _” add(“_.symbol,F.v,id=new_symbol()”),
          F.v=id} },  E>
```

Generation of tuples in bottom-up analysis

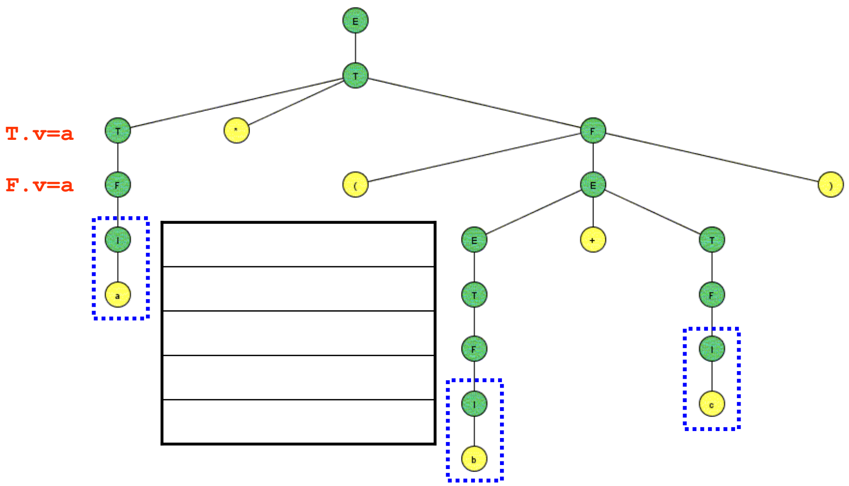
Example

- The quadruples are generated correctly:



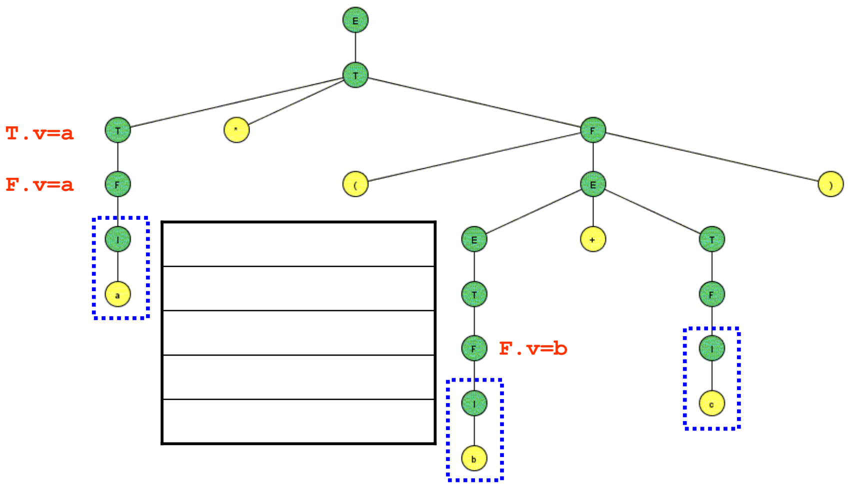
Generation of tuples in bottom-up analysis

Example



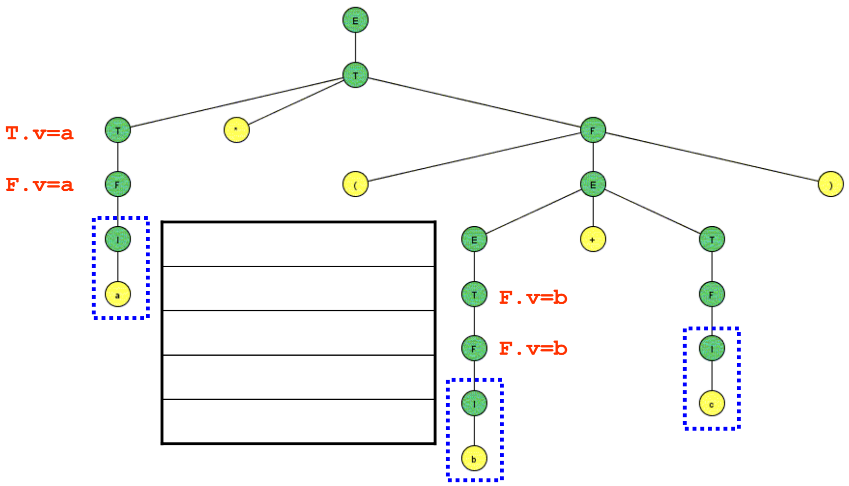
Generation of tuples in bottom-up analysis

Example



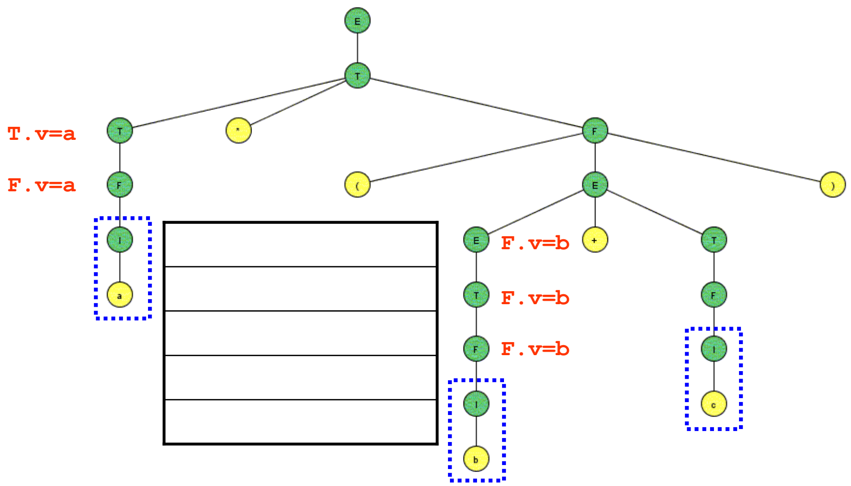
Generation of tuples in bottom-up analysis

Example



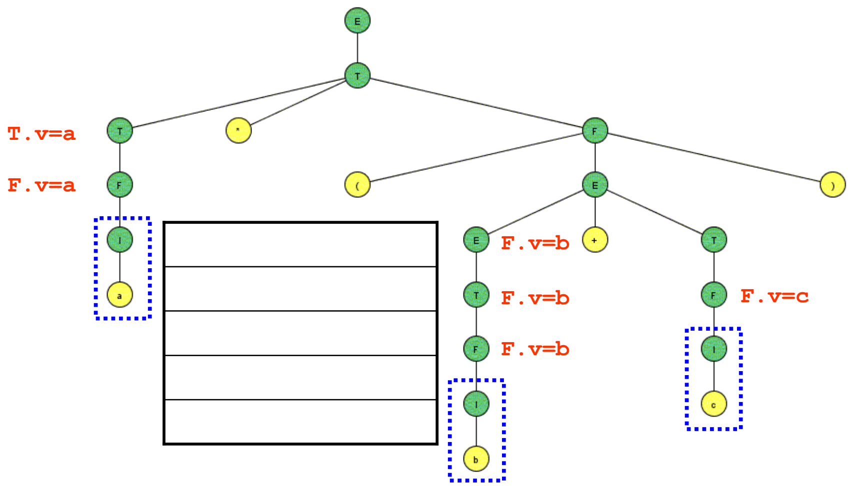
Generation of tuples in bottom-up analysis

Example



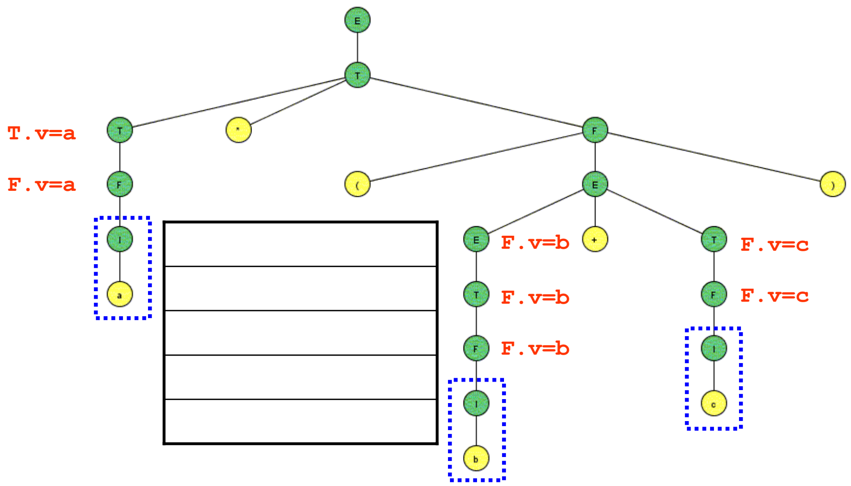
Generation of tuples in bottom-up analysis

Example



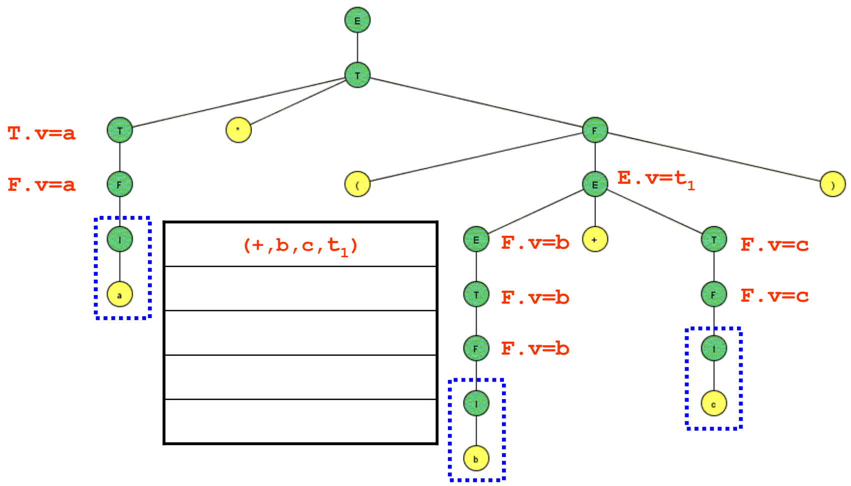
Generation of tuples in bottom-up analysis

Example



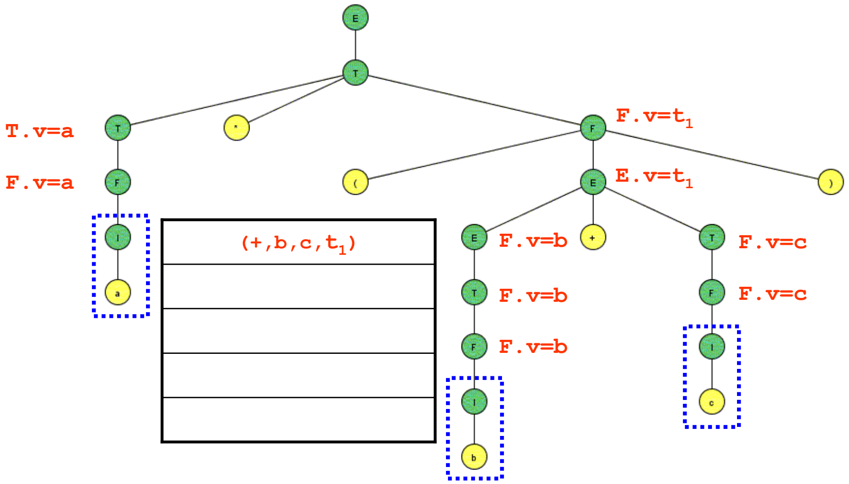
Generation of tuples in bottom-up analysis

Example



Generation of tuples in bottom-up analysis

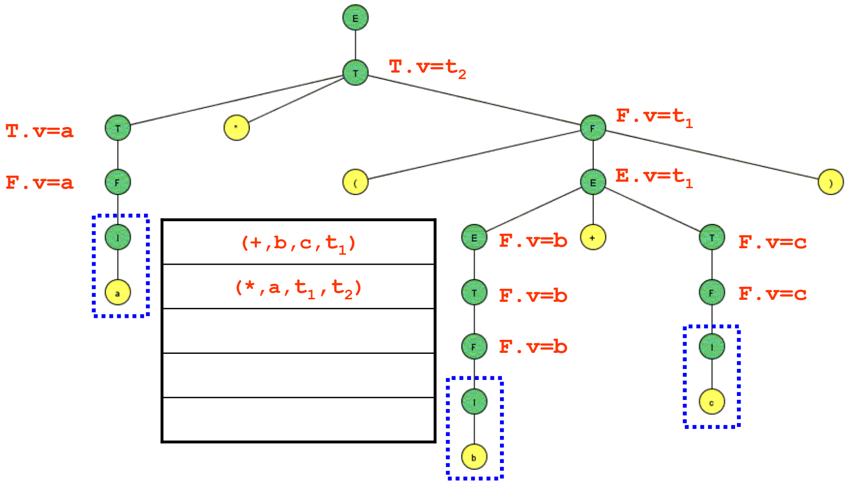
Example



(+,b,c,t_1)

Generation of tuples in bottom-up analysis

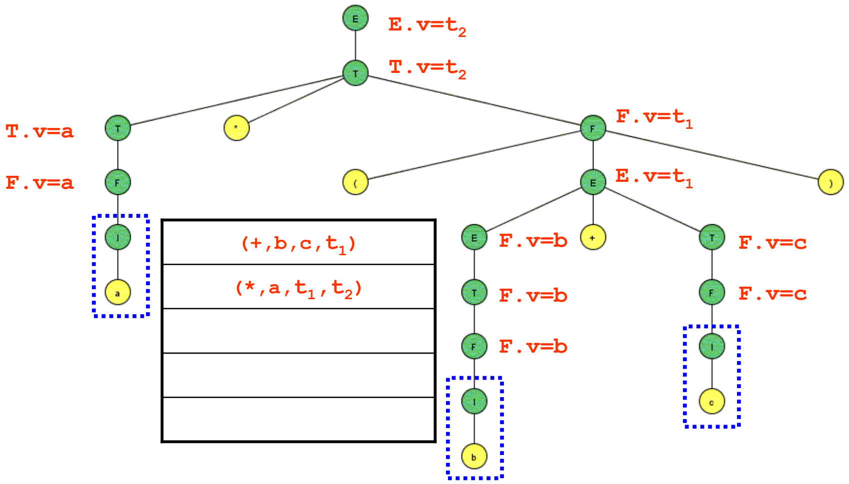
Example



(+,b,c,t_1)
(*,a,t_1,t_2)

Generation of tuples in bottom-up analysis

Example



(+,b,c,t_1)
(*,a,t_1,t_2)

Generation of tuples in top-down analysis

Example

- Concerning top-down analysis, we can also generate the tuples during the process.
- We saw the following grammar when we studied LL(1) parsers:

G_3 =<{E,T,F},{i,+,-,*,/,(,)}
{E → T+E | T-E | T
T → F*T | F/T | F
F → i | (E)
},
E>

Generation of tuples in selective top-down analysis

Example

- It had the following LL(1) equivalent grammar:

```
G3 =<{E,T,F,M,S,P,D,C},
    {i,+,-,*,/,(,)}
  {E→ iV  | (ECV
   V→ *TX | /TX | +E | -E | λ
   X→ +E  | -E  | λ
   T→ iU  | (ECU
   U→ *T  | /T  | λ
   F→ i   | (EC
   C→ )},  E>
```

Generation of tuples in selective top-down analysis

Example

- The following was the code for the LL(1) parser:

```
int E(char * string, int i)
{
// Propagate previous errors
if (i < 0) return i;

 switch (string[i]) {
   case ’i’:
     i++;
     i=V(string, i);
     break;
   case `(`:
     i++;
     i=E(string, i);
     i=C(string, i);
     i=V(string, i);
     break;
   default: return -1;/*no λ*/
  }
 return i;
}
```

```
int V(char * string, int i)
{
// Propagate previous errors
if (i < 0) return i;

switch (string[i]) {
   case ’*’:
   case ’/’:
     i++;
     i=T(string, i);
     i=X(string, i);
     break;
   case `+`:
   case `-`:
     i++;
     i=E(string, i);
     break; /* λ */
  }
 return i;
}
```

Generation of tuples in selective top-down analysis

Example

```
int X(char * string, int i)
{
// Propagate previous errors
if (i < 0) return i;

switch (string[i]) {
   case ’+’:
   case ’-’:
     i++;
     i=E(string, i);
     break; /* λ */
 }
 return i;
}
```

```
int T(char * string, int i)
{
// Propagate previous errors
if (i < 0) return i;

switch (string[i]) {
   case ’i’:
     i++;
     i=U(string, i);
     break;
   case `(`:
     i++;
     i=E(string, i);
     i=C(string, i);
     i=U(string, i);
     break;
   default: return -2;/*no λ*/
  }
 return i;
}
```

Generation of tuples in selective top-down analysis

Example

```
int U(char * string, int i)
{
// Propagate previous errors
if (i < 0) return i;

 switch (string[i]) {
   case ’*’:
   case ’/’:
     i++;
     i=T(string, i);
     break; /* λ */
 }
 return i;
}
```

```
int F(char * string, int i)
{
// Propagate previous errors
if (i < 0) return i;

 switch (string[i]) {
   case ’i’:
     i++;
     break;
   case `(`:
     i++;
     i=E(string, i);
     i=C(string, i);
     break;
   default: return -3;/*no λ*/
  }
 return i;
}
```

Generation of tuples in selective top-down analysis

Example

```
int C(char * string, int i)
{
// Propagate previous errors
if (i < 0) return i;

 switch (string[i]) {
   case ’)’:
     i++;
     break;
   default: return -4;/*no λ*/
 }
 return i;
}
```

Generation of tuples in selective top-down analysis

Example

- The string x is analysed with the following function call

```
axioma( x, 0);
```

- If the return value is the length of the input string, then it was considered correct. Otherwise, that value will contain the negated number of the rule where the error was detected.
- Execution example:

```
E(“i + i * i”, 0)
 0[1]: E( “I + I * I”, 0)
   1[1]: V( “I + I * I”, 1)
     2[1]: E( “I + I * I”, 2)
       3[1]: V( “I + I * I”, 3)
         4[1]: T( “I + I * I”, 4)
           5[1]: U( “I + I * I”, 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( “I + I * I”, 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```

Generation of tuples in selective top-down analysis

Example

- Incorrect string:

```
E(“i + i *”, 0)
 0[1]: E( “I + I *”, 0)
   1[1]: V( “I + I *”, 1)
     2[1]: E( “I + I *”, 2)
       3[1]: V( “I + I *”, 3)
         4[1]: T( “I + I *”, 4)
         4[1]: returned -2
         4[1]: X( “I + I *”, -2)
         4[1]: returned -2
       3[1]: returned -2
     2[1]: returned -2
   1[1]: returned -2
 0[1]: returned -2
-2
```

Generation of tuples in selective top-down analysis

Example

- The tuples can be generated in the middle of the code (in ***red***) as follows:

```
int E(char * string, int i)
{
// Propagate previous errors
 if (i < 0) return i;

switch (string[i]) {
   case ’i’:
     push(i.name);
     i++;
     i=V(string, i);
     break;
   case ‘(‘:
     i++;
     i=E(string, i);
     i=C(string, i);
     i=V(string, i);
     break;
   default: return -1;/*no λ*/
  }
 return i;
}
```

```
int V(char * string, int i)
{int j; char Ti[255];
// Propagate previous errors
 if (i < 0) return i;

 switch (string[i]) {
   case ’*’:
   case ’/’:
     j=i;
     i++;
     i=T(string, i);
     quad(string[j],pop(),pop(),gen(Ti));
     push(Ti);
     i=X(string, i);
     break;
   case ‘+‘:
   case ‘-‘:
     j=i;
     i++;
     i=E(string, i);
     quad(string[j],pop(),pop(),gen(Ti));
     push(Ti);
     break; /* λ */}
 return i;}
```

Generation of tuples in selective top-down analysis

Example

```
int X(char * string, int i)
{int j; char Ti[255];
// Propagate previous errors
 if (i < 0) return i;

switch (string[i]) {
   case ’+’:
   case ’-’:
     j=i;
     i++;
     i=E(string, i);
     quad(string[j],pop(),pop(),gen(Ti));
     push(Ti);
     break; /* λ */
 }
 return i;
}
```

```
int T(char * string, int i)
{
// Propagate previous errors
 if (i < 0) return i;


 switch (string[i]) {
   case ’i’:
     push(i.name);
     i++;
     i=U(string, i);
     break;
   case ‘(‘:
     i++;
     i=E(string, i);
     i=C(string, i);
     i=U(string, i);
     break;
   default: return -2;/*no λ*/
  }
 return i;
}
```

Generation of tuples in selective top-down analysis

Example

```
int U(char * string, int i)
{int j; char Ti[255];
// Propagate previous errors
 if (i < 0) return i;

switch (string[i]) {
   case ’*’:
   case ’/’:
     i++;
     i=T(string, i);
     quad(string[j],pop(),pop(),gen(Ti));
     push(Ti);
     break; /* λ */
 }
 return i;
}
```

```
int F(char * string, int i)
{
// Propagate previous errors
 if (i < 0) return i;

 switch (string[i]) {
   case ’i’:
     push(i.name);
     i++;
     break;
   case ‘(‘:
     i++;
     i=E(string, i);
     i=C(string, i);
     break;
   default: return -3;/*no λ*/
  }
 return i;
}
```

Generation of tuples in selective top-down analysis

Example

```
int C(char * string, int i)
{
// Propagate previous errors
 if (i < 0) return i;

 switch (string[i]) {
   case ’)’:
     i++;
     break;
   default: return -4;/*no λ*/
 }
 return i;
}
```

Generation of tuples in selective top-down analysis

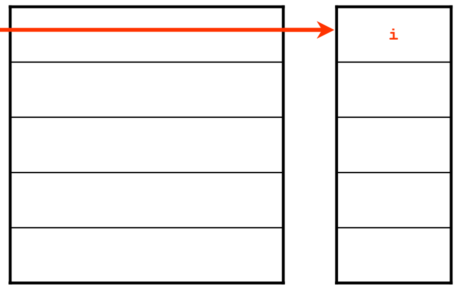
Example

- The string `x` is analysed with the following call:

```
axiom( x, 0);
```

- If the return value is the length of the input string, then it was considered correct. Otherwise, that value will contain the negated number of the rule where the error was detected.
- Execution examples:

```
E(“i + i * i”, 0)
 0[1]: E( “I + I * I”, 0)
   1[1]: V( “I + I * I”, 1)
     2[1]: E( “I + I * I”, 2)
       3[1]: V( “I + I * I”, 3)
         4[1]: T( “I + I * I”, 4)
           5[1]: U( “I + I * I”, 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( “I + I * I”, 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```

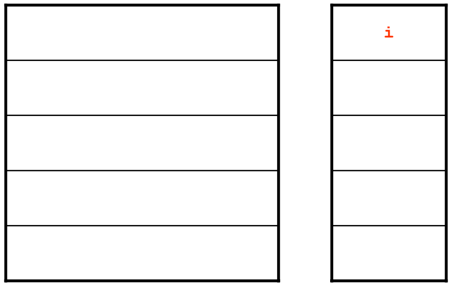


Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```

```
E(“i + i * i”, 0)
 0[1]: E( “I + I * I”, 0)
   1[1]: V( “I + I * I”, 1)
     2[1]: E( “I + I * I”, 2)
       3[1]: V( “I + I * I”, 3)
         4[1]: T( “I + I * I”, 4)
           5[1]: U( “I + I * I”, 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( “I + I * I”, 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```

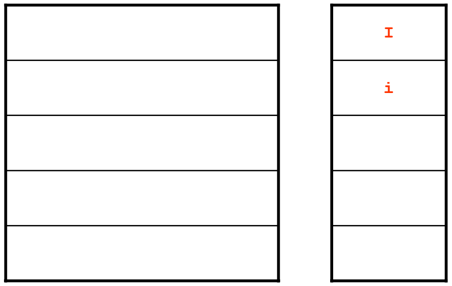


Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```

```
E(“i + i * i”, 0)
 0[1]: E( “I + I * I”, 0)
   1[1]: V( “I + I * I”, 1)
     2[1]: E( “I + I * I”, 2)
       3[1]: V( “I + I * I”, 3)
         4[1]: T( “I + I * I”, 4)
           5[1]: U( “I + I * I”, 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( “I + I * I”, 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```

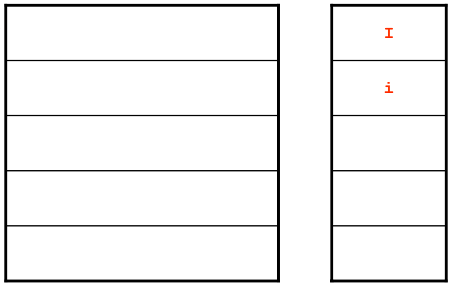


Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```

```
E(“i + i * i”, 0)
 0[1]: E( “I + I * I”, 0)
   1[1]: V( “I + I * I”, 1)
     2[1]: E( “I + I * I”, 2)
       3[1]: V( “I + I * I”, 3)
         4[1]: T( “I + I * I”, 4)
           5[1]: U( “I + I * I”, 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( “I + I * I”, 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```

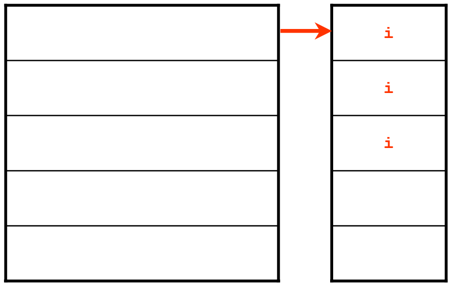


Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```

```
E(“i + i * i”, 0)
 0[1]: E( “I + I * I”, 0)
   1[1]: V( “I + I * I”, 1)
     2[1]: E( “I + I * I”, 2)
       3[1]: V( “I + I * I”, 3)
         4[1]: T( “I + I * I”, 4)
           5[1]: U( “I + I * I”, 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( “I + I * I”, 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```

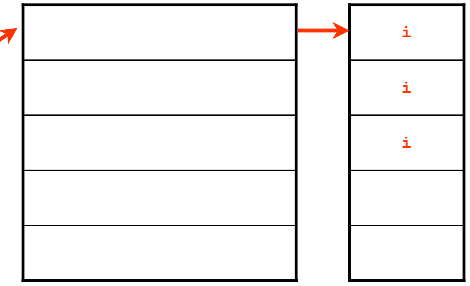


Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```

```
E(“i + i * i”, 0)
 0[1]: E( “I + I * I”, 0)
   1[1]: V( “I + I * I”, 1)
     2[1]: E( “I + I * I”, 2)
       3[1]: V( “I + I * I”, 3)
         4[1]: T( “I + I * I”, 4)
           5[1]: U( “I + I * I”, 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( “I + I * I”, 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```

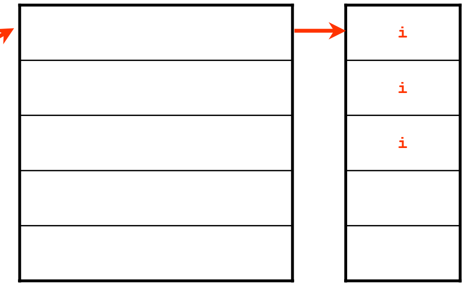


Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```

```
E(“i + i * i”, 0)
 0[1]: E( “I + I * I”, 0)
   1[1]: V( “I + I * I”, 1)
     2[1]: E( “I + I * I”, 2)
       3[1]: V( “I + I * I”, 3)
         4[1]: T( “I + I * I”, 4)
           5[1]: U( “I + I * I”, 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( “I + I * I”, 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```

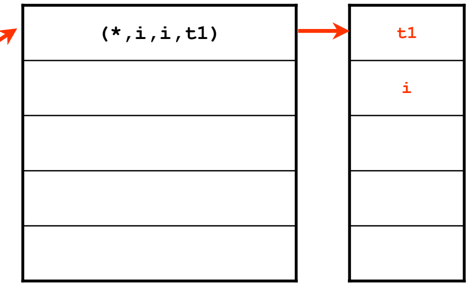


Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```

```
E(“i + i * i”, 0)
 0[1]: E( “I + I * I”, 0)
   1[1]: V( “I + I * I”, 1)
     2[1]: E( “I + I * I”, 2)
       3[1]: V( “I + I * I”, 3)
         4[1]: T( “I + I * I”, 4)
           5[1]: U( “I + I * I”, 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( “I + I * I”, 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```

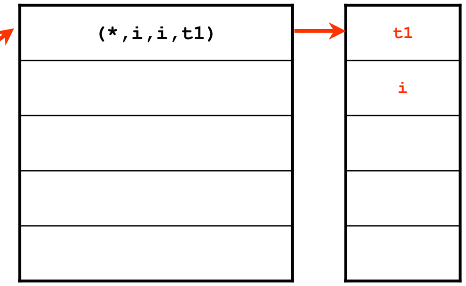


Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```

```
E(“i + i * i”, 0)
 0[1]: E( “I + I * I”, 0)
   1[1]: V( “I + I * I”, 1)
     2[1]: E( “I + I * I”, 2)
       3[1]: V( “I + I * I”, 3)
         4[1]: T( “I + I * I”, 4)
           5[1]: U( “I + I * I”, 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( “I + I * I”, 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```

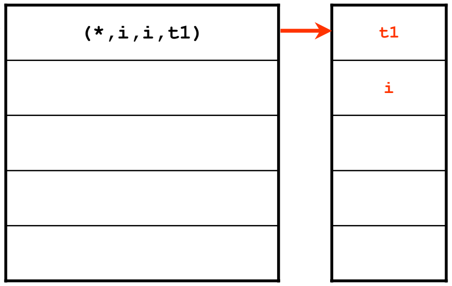


Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```

```
E("i + i * i", 0)
 0[1]: E( "I + I * I", 0)
   1[1]: V( "I + I * I", 1)
     2[1]: E( "I + I * I", 2)
       3[1]: V( "I + I * I", 3)
         4[1]: T( "I + I * I", 4)
           5[1]: U( "I + I * I", 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( "I + I * I", 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```


(*,i,i,t1)

t1

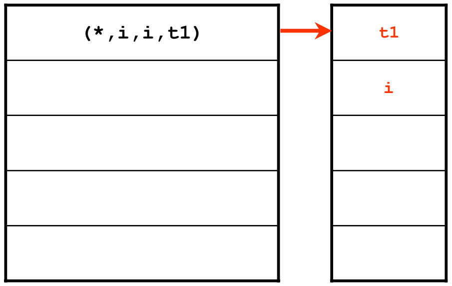
i

Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```

```
E("i + i * i", 0)
 0[1]: E( "I + I * I", 0)
   1[1]: V( "I + I * I", 1)
     2[1]: E( "I + I * I", 2)
       3[1]: V( "I + I * I", 3)
         4[1]: T( "I + I * I", 4)
           5[1]: U( "I + I * I", 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( "I + I * I", 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```


(*,i,i,t1)

t1

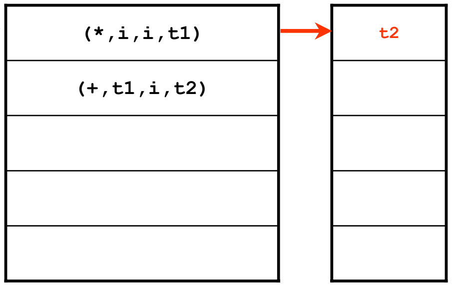
i

Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```

```
E("i + i * i", 0)
 0[1]: E( "I + I * I", 0)
   1[1]: V( "I + I * I", 1)
     2[1]: E( "I + I * I", 2)
       3[1]: V( "I + I * I", 3)
         4[1]: T( "I + I * I", 4)
           5[1]: U( "I + I * I", 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( "I + I * I", 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```


(*,i,i,t1)

(+,t1,i,t2)

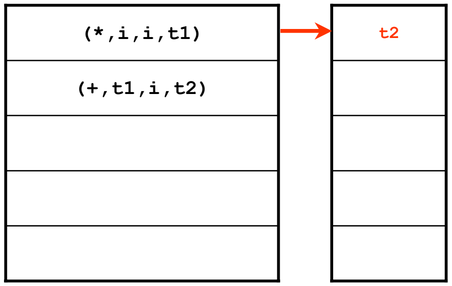
t2

Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```

```
E("i + i * i", 0)
 0[1]: E( "I + I * I", 0)
   1[1]: V( "I + I * I", 1)
     2[1]: E( "I + I * I", 2)
       3[1]: V( "I + I * I", 3)
         4[1]: T( "I + I * I", 4)
           5[1]: U( "I + I * I", 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( "I + I * I", 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```


(*,i,i,t1)

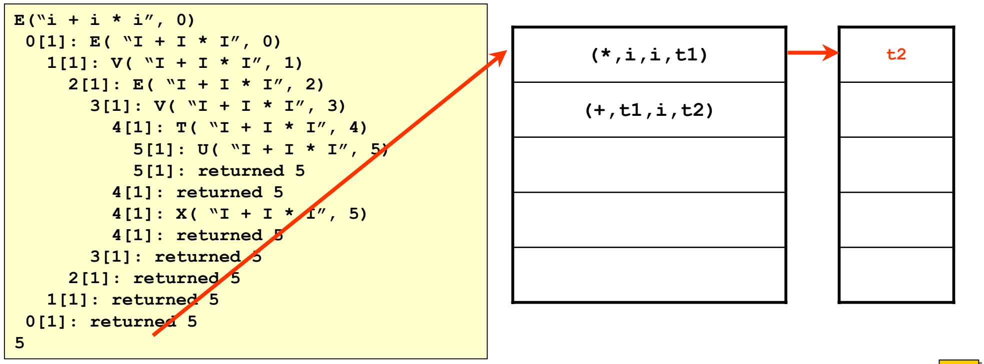
(+,t1,i,t2)

t2

Generation of tuples in selective top-down analysis

Example

```
axiom( x, 0);
```



```
E("i + i * i", 0)
 0[1]: E( "I + I * I", 0)
   1[1]: V( "I + I * I", 1)
     2[1]: E( "I + I * I", 2)
       3[1]: V( "I + I * I", 3)
         4[1]: T( "I + I * I", 4)
           5[1]: U( "I + I * I", 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( "I + I * I", 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```

(*,i,i,t1)

(+,t1,i,t2)

t2

Tuples

General concepts

- Most of the assembly languages use operation codes with three addresses. In other words, each instruction contains:
  - The operation to perform.
  - At most two operands (all operations are considered dyadic or monadic)
  - The place there the result will be placed (the name of a variable, a memory address, or a register)
- Tuples ensure an easy translation into assembly language.

Tuples

General concepts: available operations

- Usually, we can find:
  - Operations to assign an expression to a variable:
    - Examples:

```
x := y <Binary_operator> z
x := <Monadic_operator> y
x := y
```

      - In these examples, **x** is assigned the result of the operation.
      - **Binary operators** include the most frequent arithmetic and logic operators.
      - **Monadic operators** are usually,
        - Negation.
        - Changing the sign of a number.
        - Change the data type.

Tuples

General concepts: available operations

- Unconditional jumps:

```
jump_to <Label>
```

- Conditional jumps:

```
<operation>
if <condition> jump_to <Label>
```

  - In this case, the condition is the last one to be evaluated, so the “state flags” will be set right before the jump operation.
- If-then-else statements:

```
if <logic_operation> then <statements> else <statementsElse>
```

  - In this case, if the operation is correct, then **<statements>** will execute. Otherwise, **<statementsElse>** will execute.
  - After any of these two options, the program follows with the next instruction.
- Other operations: accessing fields in structures and cells in matrices, passings arguments to functions, function calls, etc.

Tuples

General concepts

- We can use different implementations for the intermediate representation of this “operation codes in three directions”.
  - **Quadruples** with the following structure:

    `(Operator, Operand1, Operand2, Result)`
  - **Triplets**, with the structure

    `(Operator, Operand1, Operand2)`
    - In the case of triplets, the triplet itself represents the result of the operation. It will be identified by its position in the ordering of triplets.

- The following slides contain some examples of generation of quadruples for the constructions that we have enumerated.

Tuples

General concepts

- We shall use the following set of quadruples:

  `(jump_if<condition>, <jump_address>,,<Operand>)`
  - for conditional jumps to **`<jump_address>`** according to the value of **`<Operand>`**

  `(jump_to, <jump_address>,,)`
  - for unconditional jumps.

  `(:=, <expression>,,<target>)`
  - For assigning the value of **`<expression>`** to the variable **`<target>`**

  `(<Dyadic_operator>, <Operand>, <Operand>, <target>)`
  - for the typical arithmetic and logic operations.

  `(<Monadic_operator>, <Operand>,, <target>)`
  - For the typical monadic operations

Quadruples

Semantics of conditional flow statements

- With respect to the conditional flow,

  - let us see the following attributes grammar:

G_3=< {<Instr>, <Expr>},
{if }
{
<Instr>→if <Expr>{S_2}then <Instr>{S_1}
<Instr>→if <Expr>{S_2}then <Instr> else{S_3}<Instr>{S_1}
},
<Instr>
>

Quadruples

Semantics of conditional flow statements

- We shall assume that the semantic analyser has the following resources:
  - A stack to store useful intermediate information for generating the quadruples, usable with the operations **push** and **pop**.
  - A matrix with quadruples, **`c`**.
  - A function

    `quadruple_label next_quadruple()`

    which returns a new label for identifying a quadruple, corresponding to the first empty position in the quadruples matrix.
  - A function

    `quadruple_no generate_tuple(tuple)`

    to insert the tuple provided as argument in the next empty position in the matrix. It returns the number of the newly inserted quadruple.
  - An auxiliary variable, with name **`id`** to store the names of the identifiers.
  - After analysing an expression, we shall leave, at the top of the stack, the identifier of the last quadruple in which we calculated the result of the expression.
  - An auxiliary variable **`id_quadruple`** to store identifiers of quadruples.

Quadruples

Semantics of conditional flow statements

```
S1={    c[top][2]=next_quadruple();
        pop;}

S2={    pop id; /* id contains the result of <Expr> */
        id_quadruple=generate_tuple(“(jump_if_false,,,”id“)”);
        push id_quadruple;}

S3={    id_quadruple=generate_tuple(“(jump_to,,,)”);
        c[top][2]=next_quadruple();
        pop;
        push id_quadruple;}


G3 =<   {<Instr>, <Expr>},  {if }
        { <Instr>→if <Expr>{S2}then <Instr>{S1}
          <Instr>→if <Expr>{S2}then <Instr> else{S3}<Instr>{S1}},
         <Instr> >
```

Quadruples: loops

Examples

- Apply the attributes grammar to the next program:

```
if ( (a+b) < (c*d))
then
  a:=a/b
else
  a:=a*b

if ( a < b )
then
  x:=1
else
  x:=2

if E1
then
  I1
```

```
1 (+, a, b, t1)
2 (*, c, d, t2)
3 (<, t1, t2, t3)
4 (jump-if-false, 7, , t3)
5 (/, a, b, a)
6 (jump-to, 8, , )
7 (*, a, b, a)
8
```

Quadruples

Semantics of Unconditional Jumps. Labels.

- The following is an example of the GOTO operation.
- It includes how to manage the labels.
- We shall work with the following attributes grammar:

```
G3 =<   {<Instr>},
        {id, goto, “:”}
        {
              <Instr>→id : <Instr> {declare_label(id);}
              <Instr>→goto id {goto_identifier(id);}
        },
         <Instr>
>
```

Quadruples

Semantics of Unconditional Jumps. Labels.

- The semantic actions are specified next. We shall assume, in the code, that the analyser has accessible:
  - The symbols table **`TS`**
  - A data type **`value`** for the values stored in the symbols table.
  - Functions to access the symbols table:
    - **`insert(identifier id, value value,table TS);`**
    - **`change(identifier id, value new_value,table TS);`**
    - **`value search(identifier id ,table TS);,`**
  - We shall assume that the data type **`value`** contains:
    - Identifier type (**`type`**): **`label`**
    - A flag to indicate whether it has been declared or not.
    - The label of the next quadruple (**`n°_quadruple`**)
  - A quadruples matrix **`c`**.

Quadruples

Semantics of Unconditional Jumps. Labels.

- A function `quadruple_label next_quadruple();` that returns a new label to identify a quadruple, corresponding to the first empty position in **c**.
- Auxiliary variables `i, j` and `new_quadruple` of the same type as the labels for identifying the quadruples (`n°_quadruple`)
- A function `generate_tuple(tuple)` that inserts the tuple provided as argument in the next position in the matrix **c**.
- A function:

  `define_label(identifier label, n°_quadruple number)`

  - that associates a label to a certain quadruple number.
- A function

  `error();`

  - that starts the treatment of an error condition.

Quadruples

Semantics of Unconditional Jumps. Labels.

```
declare_label(id){
  if ( (value=search(id,TS)) == NOT_FOUND ){
        new_quadruple = next_quadruple();
        insert(id,("label", "declared", new_quadruple),TS);
        define_label(id,new_quadruple);
  }
  else {
        if (value.type == "label" AND value.declared == "no"){
                i=value.n°_quadruple;
                new_quadruple=next_quadruple();
                while ( i != "empty quadruple"){
                        j=c[i][2];
                        c[i][2]=new_quadruple;
                        i=j;
                }
                change(id,("label","declared",new_quadruple),TS);
                define_label(id, new_quadruple);
        }
        else error();
  }
}
```

Quadruples

Semantics of Unconditional Jumps. Labels.

```
goto_identifier(id){
  if ( (value = search(id,TS)) == NOT_FOUND ){
    insert(id,("label","undeclared", next_quadruple()),TS);
    generate_tuple("(jump_to,,,)");
   }
  else {
    if (value.type == "label") {
     if (value.declared == "YES") {
       generate_tuple("(jump_to,"value.quadruple_number",,)");}
      else if (value.declared == "NO"){
       i = value.quadruple_number;
       change(id,("label","undeclared",next_quadruple()),TS );
       generate_tuple("(jump_to," i ",,)");
      }
     }
    else error();
   }
}
```

Quadruples: loops

Examples

- Apply the attributes grammar to the next case:

```
     goto L1
     ...
     goto L2
     ...
  L1:...
     ...
  L2:...
     goto L1
     ...
     goto L2
     ...
```

```
insert(L1,("undeclared", 1))
insert(L2,("undeclared", n1))


1  (jump_to, n2, , )
...
n1  (jump_to, n3, , )
...
n2  (...)
...
n3  (...)
```

Quadruples

Semantics of iterative control flow.

- The following is another example for generating quadruples for a looping statement. It shall be used to perform something a given number of times, depending on the initial value of a variable and a given increment.
- The following is the attributes grammar:

```
G3 =<   {<Instr>, <Expr>, <CD1>, <Linstr>},
        {do, id, :=, ",″}
        {
                <Instr>→
                  do id {S0}:=<Expr>1 {S1},
                              <Expr>2 {S2} <CD1>
                     <Linstr>
                 end {S5},
                 <CD1>→<Expr>3{S3}
                 <CD1>→λ{S4}
                 <Linstr>→″Sequences of <Instr> separated by ; "
                 <Expr>→″Arithmetic expressions"
        },
         <Instr>
>
```

Quadruples

Semantics of iterative control flow.

- For the semantic actions, we are going to assume that we have now...
  - A **`stack`** to store all the useful intermediate information to generate the quadruples, with the operations **push** and **pop**.
  - Whenever a expression is processed, we shall assume that we have, at the top of the stack, the identifier where the result is, or the value of the result.
  - A matrix with quadruples, called **`c`**.
  - Auxiliary variables **`i`** and **`j`** with the same type as the labels of the quadruples.
  - An auxiliary variable **`exp`** to store the value of arithmetic expressions.
  - An auxiliary variable **`id`** to store the names of the identifiers.
  - A function **`quadruple_label next_quadruple();`** that returns a new label to identify a quadruple, corresponding to the first empty position in matrix **c**.
  - A function **`generate_tuple(tuple)`** to insert, in the next empty position of the matrix **c**, the tuple provided as argument.

Quadruples

Semantics of iterative control flow.

```
S0={push id;}
S1={    pop exp;
        pop id;
        generate_tuple("(:=,″ ,exp,  ″, ,″ ,id ″)″;
        i:= next_quadruple();
        push id;
    }

S2={    pop exp;
        pop id;
        j:= next_quadruple();
        generate_tuple("(jump_if_greater,,″ ,id, ",″,  exp ")″);
        generate_tuple("(jump_to,,,)″);
        push id;
    }
```

Quadruples

Semantics of iterative control flow.

```
S3={    pop exp;
        pop id;
        generate_tuple( "(+,″,id ",″ ,ex ",″ ,id  ")″);
        generate_tuple( "(jump_to,″ , i, ",,)″ );
        c[j+1][2]:=next_quadruple();
   }
S4={    pop id;
        generate_tuple("(+,″ ,id, ″,1,″ ,id ″)″);
        generate_tuple("(jump_to,″ ,i, ",,)″);
        c[j+1][2]:=next_quadruple();
    }
S5={    generate_tuple( "(jump_to,″ ,j+2, ",,)″);
        c[j][2]:=next_quadruple();
    }
```

Quadruples: loops

Examples

- Apply the previous attributes grammar to the following two cases.
- We are only interested in the quadruples related to the loop. The quadruples for the expressions and assignments are not relevant in this example.

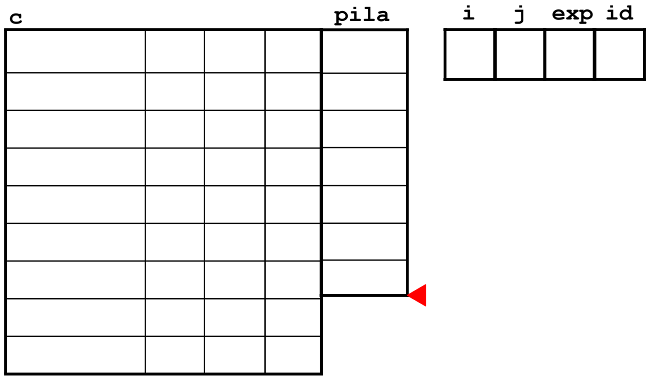
```
do x:=1, 10
  z:=1;
  y:=2
end
```

```
do x:=1, 10, 2
  z:=1;
  y:=2
end
```

Quadruples: loops

Examples

```
do x:=1, 10
  z:=1;
  y:=2
end
```

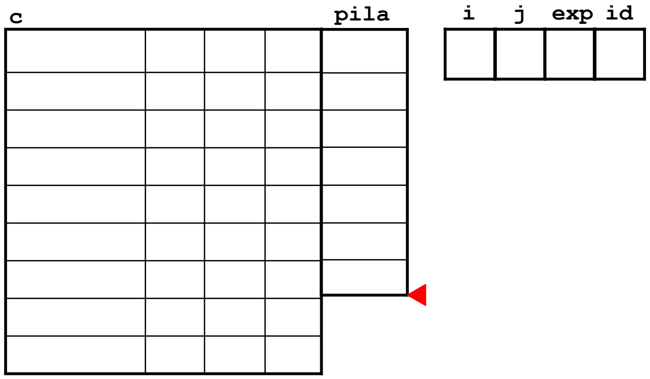


Quadruples: loops

Examples

do x {S_0}:=1, 10
  z:=1;
  y:=2
end

S_0={push id;}

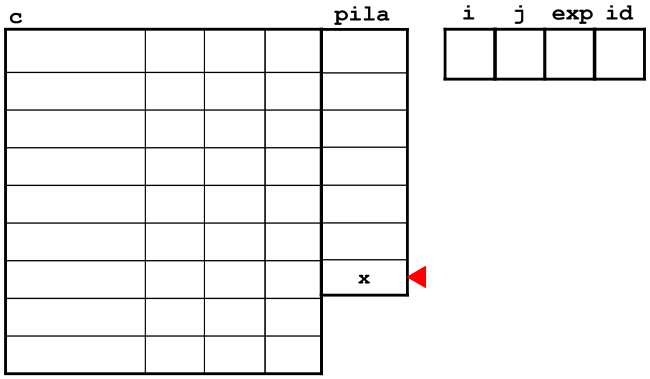


Quadruples: loops

Examples

do x {S_0}:=1, 10
  z:=1;
  y:=2
end

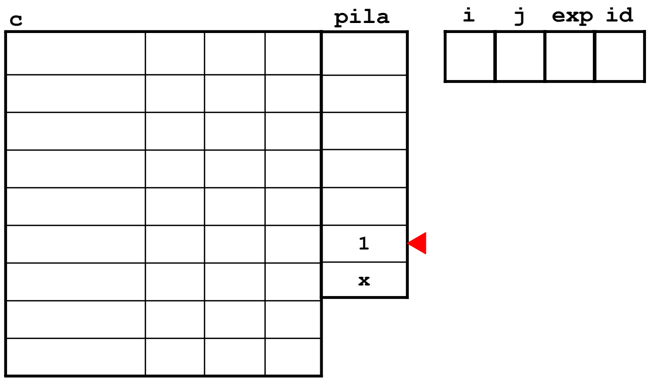
S_0={push id;}



Quadruples: loops

Examples

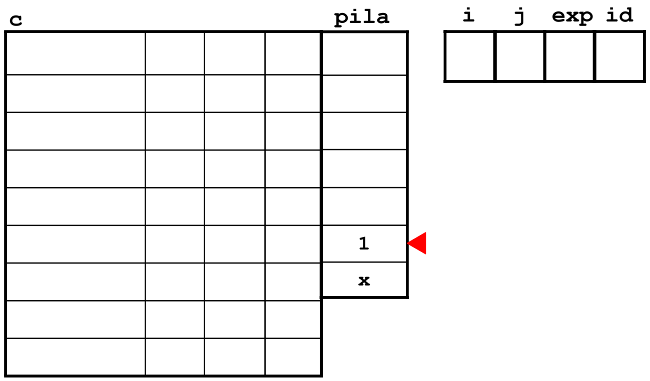
```
do x {$S_0$}:=1, 10
  z:=1;
  y:=2
end
```



Quadruples: loops

Examples

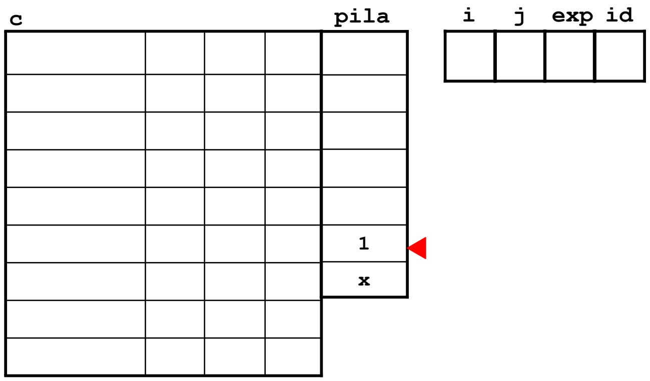
```
do x {$S_0$}:=1{$S_1$}, 10
  z:=1;
  y:=2
end
```



Quadruples: loops

Examples

```
do x {$S_0$}:=1{$S_1$}, 10
  z:=1;
  y:=2
end
```

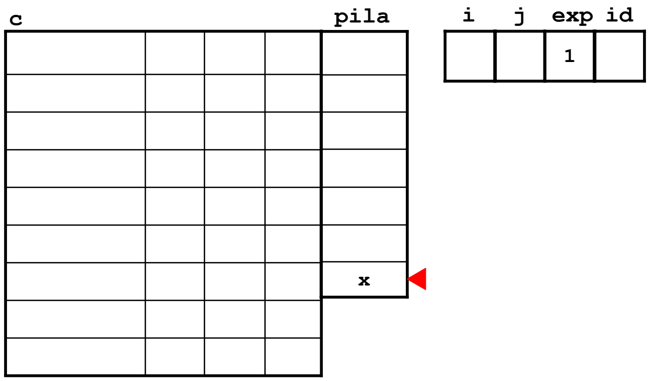


```
$S_1$={
 pop exp;
 pop id;
 generate_tuple(
       "(:=,",exp,
       ", ," ,id ")";
 i:= next_quadruple();
 push id;
 }
```

Quadruples: loops

Examples

```
do x {$S_0$}:=1{$S_1$}, 10
  z:=1;
  y:=2
end
```



```
$S_1$={
 pop exp;
 pop id;
 generate_tuple(
       "(:=,",exp,
       ", ," ,id ")";
 i:= next_quadruple();
 push id;
 }
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10
   z:=1;
   y:=2
end
```

c

1	:=	1		x

pila

i	j	exp	id
		1	x

```
S1={
 pop exp;
 pop id;
 generate_tuple(
        "(:=,",exp,
        ",  ," ,id ")";
 i:= next_quadruple();
 push id;
 }
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10
   z:=1;
   y:=2
end
```

c

1	:=	1		x

pila

i	j	exp	id
2		1	x

```
S1={
 pop exp;
 pop id;
 generate_tuple(
        "(:=,",exp,
        ",  ," ,id ")";
 i:= next_quadruple();
 push id;
 }
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10
   z:=1;
   y:=2
end
```

c

1	:=	1		x

pila
x

i	j	exp	id
2		1	x

```
S1={
 pop exp;
 pop id;
 generate_tuple(
        "(:=,",exp,
        ",  ," ,id ")";
 i:= next_quadruple();
 push id;
 }
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10
   z:=1;
   y:=2
end
```

c

1	:=	1		x

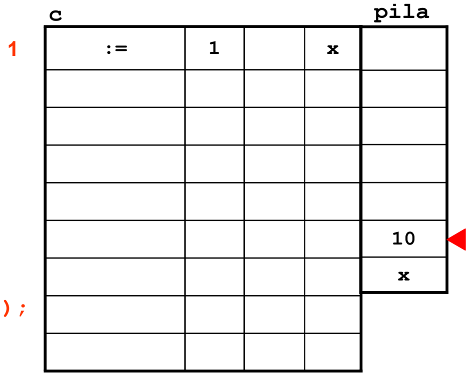
pila
10
x

i	j	exp	id
2		1	x

Quadruples: loops

Examples

```
  do x {S0}:=1{S1}, 10{S2}
    z:=1;
    y:=2
  end
```



	c				pila
1	:=	1		x	
					10
					x

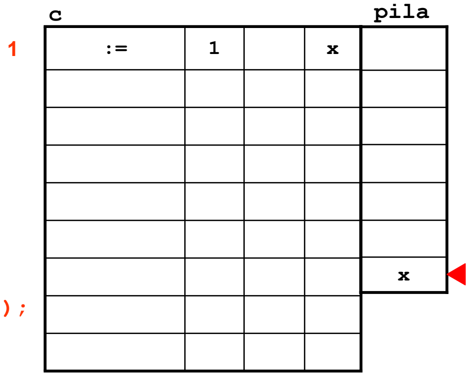
i	j	exp	id
2		1	x

```
S2={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    “(jump_if_greater,,” ,id, “,”,  exp
“)”);
 generate_tuple(“(jump_to,,,)”);
 push id;
}
```

Quadruples: loops

Examples

```
  do x {S0}:=1{S1}, 10{S2}
    z:=1;
    y:=2
  end
```



	c				pila
1	:=	1		x	
					x

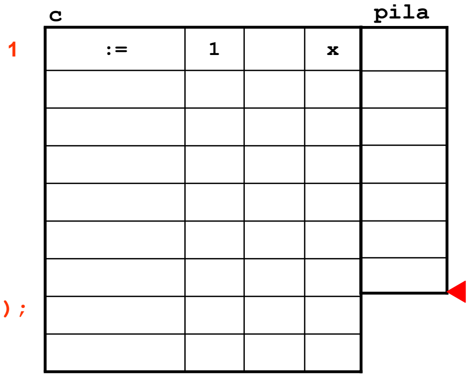
i	j	exp	id
2		10	x

```
S2={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    “(jump_if_greater,,” ,id, “,”,  exp
“)”);
 generate_tuple(“(jump_to,,,)”);
 push id;
}
```

Quadruples: loops

Examples

```
  do x {S0}:=1{S1}, 10{S2}
    z:=1;
    y:=2
  end
```



	c				pila
1	:=	1		x	

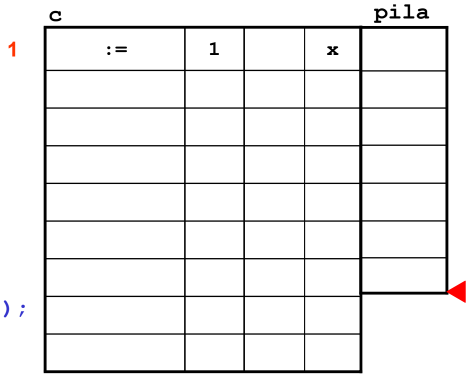
i	j	exp	id
2		10	x

```
S2={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    “(jump_if_greater,,” ,id, “,”,  exp
“)”);
 generate_tuple(“(jump_to,,,)”);
 push id;
}
```

Quadruples: loops

Examples

```
  do x {S0}:=1{S1}, 10{S2}
    z:=1;
    y:=2
  end
```



	c				pila
1	:=	1		x	

i	j	exp	id
2	2	10	x

```
S2={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    “(jump_if_greater,,” ,id, “,”,  exp
“)”);
 generate_tuple(“(jump_to,,,)”);
 push id;
}
```

Quadruples: loops

Examples

```
do x {$S_0$}:=1{$S_1$}, 10{$S_2$}
  z:=1;
  y:=2
end
```

	c			
1	:=	1		x
2	jump_>		x	10

pila

i	j	exp	id
2	2	10	x

```
$S_2$={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    “(jump_if_greater,,” ,id, “,”,  exp
“)”);
 generate_tuple(“(jump_to,,,)”);
 push id;
}
```

Quadruples: loops

Examples

```
do x {$S_0$}:=1{$S_1$}, 10{$S_2$}
  z:=1;
  y:=2
end
```

	c			
1	:=	1		x
2	jump_>		x	10
3	jump_to			

pila

i	j	exp	id
2	2	10	x

```
$S_2$={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    “(jump_if_greater,,” ,id, “,”,  exp
“)”);
 generate_tuple(“(jump_to,,,)”);
 push id;
}
```

Quadruples: loops

Examples

```
do x {$S_0$}:=1{$S_1$}, 10{$S_2$}
  z:=1;
  y:=2
end
```

	c			
1	:=	1		x
2	jump_>		x	10
3	jump_to			

pila
x

i	j	exp	id
2	2	10	x

```
$S_2$={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    “(jump_if_greater,,” ,id, “,”,  exp
“)”);
 generate_tuple(“(jump_to,,,)”);
 push id;
}
```

Quadruples: loops

Examples

```
do x {$S_0$}:=1{$S_1$}, 10{$S_2$} {$S_4$}
  z:=1;
  y:=2
end
```

	c			
1	:=	1		x
2	jump_>		x	10
3	jump_to			

pila
x

i	j	exp	id
2	2	10	x

```
$S_4$={
 pop id;
 generate_tuple(“(+,” ,id, ”,1,” ,id ”)”);
 generate_tuple(“(jump_to,” ,i, “,,)”);
 c[j+1][2]:=next_quadruple();
}
```

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2} {S_4}
z:=1;
y:=2
end

	c				pila
1	:=	1		x	
2	jump_>		x	10	
3	jump_to				

i	j	exp	id
2	2	10	x

S_4={
pop id;
generate_tuple(“(+,” ,id, ”,1,” ,id ”)”);
generate_tuple(“(jump_to,” ,i, “,,)”);
c[j+1][2]:=next_quadruple();
}

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2} {S_4}
z:=1;
y:=2
end

	c				pila
1	:=	1		x	
2	jump_>		x	10	
3	jump_to				
4	+	x	1	x	

i	j	exp	id
2	2	10	x

S_4={
pop id;
generate_tuple(“(+,” ,id, ”,1,” ,id ”)”);
generate_tuple(“(jump_to,” ,i, “,,)”);
c[j+1][2]:=next_quadruple();
}

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2} {S_4}
z:=1;
y:=2
end

	c				pila
1	:=	1		x	
2	jump_>		x	10	
3	jump_to				
4	+	x	1	x	
5	jump_to	2			

i	j	exp	id
2	2	10	x

S_4={
pop id;
generate_tuple(“(+,” ,id, ”,1,” ,id ”)”);
generate_tuple(“(jump_to,” ,i, “,,)”);
c[j+1][2]:=next_quadruple();
}

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2} {S_4}
z:=1;
y:=2
end

	c				pila
1	:=	1		x	
2	jump_>		x	10	
3	jump_to	6			
4	+	x	1	x	
5	jump_to	2			

i	j	exp	id
2	2	10	x

S_4={
pop id;
generate_tuple(“(+,” ,id, ”,1,” ,id ”)”);
generate_tuple(“(jump_to,” ,i, “,,)”);
c[j+1][2]:=next_quadruple();
}

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2} {S_4}
  z:=1;
  y:=2
end

	c				pila
1	:=	1		x	
2	jump_>		x	10	
3	jump_to	6			
4	+	x	1	x	
5	jump_to	2			
6	:=	1		z	

i	j	exp	id
2	2	10	x

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2} {S_4}
  z:=1;
  y:=2
end

	c				pila
1	:=	1		x	
2	jump_>		x	10	
3	jump_to	6			
4	+	x	1	x	
5	jump_to	2			
6	:=	1		z	
7	:=	2		y	

i	j	exp	id
2	2	10	x

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2} {S_4}
  z:=1;
  y:=2
end {S_5}

	c				pila
1	:=	1		x	
2	jump_>		x	10	
3	jump_to	6			
4	+	x	1	x	
5	jump_to	2			
6	:=	1		z	
7	:=	2		y	

i	j	exp	id
2	2	10	x

S_5={
  generate_tuple("(jump_to," ,j+2, ",,)");
  c[j][2]:=next_quadruple();
}

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2} {S_4}
  z:=1;
  y:=2
end {S_5}

	c				pila
1	:=	1		x	
2	jump_>		x	10	
3	jump_to	6			
4	+	x	1	x	
5	jump_to	2			
6	:=	1		z	
7	:=	2		y	
8	jump_to	4			

i	j	exp	id
2	2	10	x

S_5={
  generate_tuple("(jump_to," ,j+2, ",,)");
  c[j][2]:=next_quadruple();
}

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2} {S_4}
  z:=1;
  y:=2
end {S_5}

	c				pila
1	:=	1		x	
2	jump_>	9	x	10	
3	jump_to	6			
4	+	x	1	x	
5	jump_to	2			
6	:=	1		z	
7	:=	2		y	
8	jump_to	4			

i	j	exp	id
2	2	10	x

S_5={
  generate_tuple(“(jump_to,” ,j+2, “,,)”);
  c[j][2]:=next_quadruple();
}

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2} {S_4}
  z:=1;
  y:=2
end {S_5}

	c				pila
1	:=	1		x	
2	jump_>	9	x	10	
3	jump_to	6			
4	+	x	1	x	
5	jump_to	2			
6	:=	1		z	
7	:=	2		y	
8	jump_to	4			

i	j	exp	id
2	2	10	x

Quadruples: loops

Examples

do x:=1, 10, 2
  z:=1;
  y:=2
end

c				pila

i	j	exp	id

Quadruples: loops

Examples

do x:=1, 10, 2
  z:=1;
  y:=2
end

c				pila

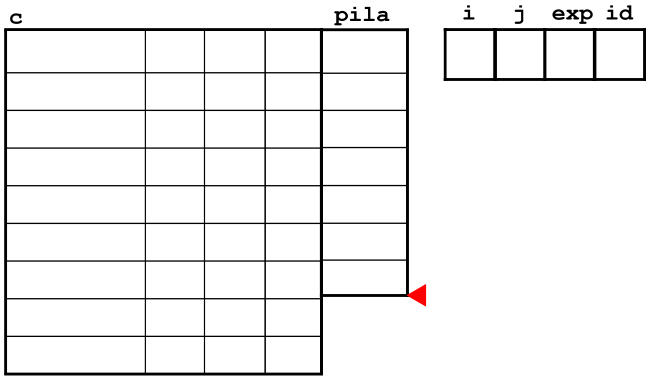
i	j	exp	id

Quadruples: loops

Examples

do x {S_0}:=1, 10 , 2
  z:=1;
  y:=2
end

S_0={push id;}

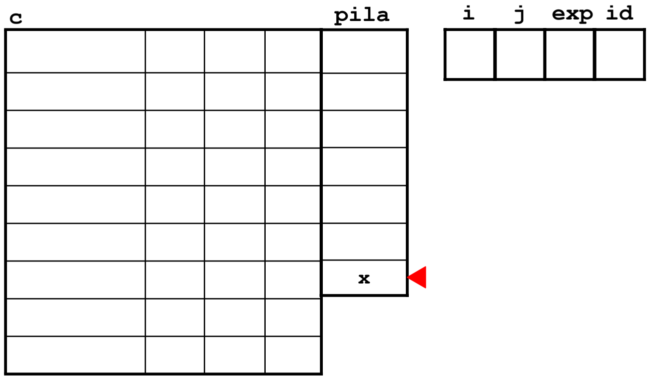


Quadruples: loops

Examples

do x {S_0}:=1, 10 , 2
  z:=1;
  y:=2
end

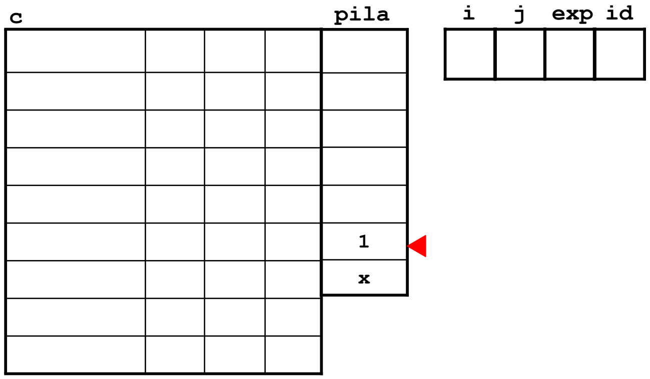
S_0={push id;}



Quadruples: loops

Examples

do x {S_0}:=1, 10, 2
  z:=1;
  y:=2
end

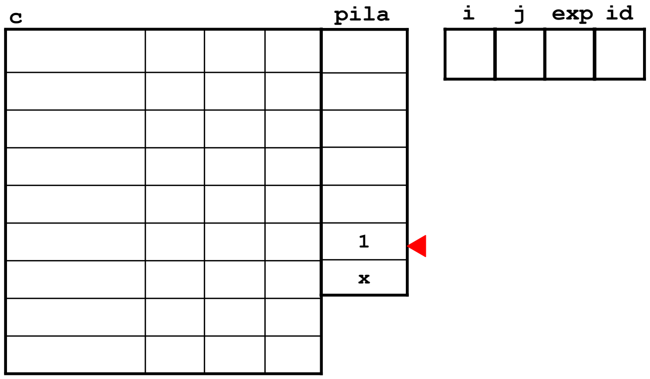


Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10, 2
  z:=1;
  y:=2
end

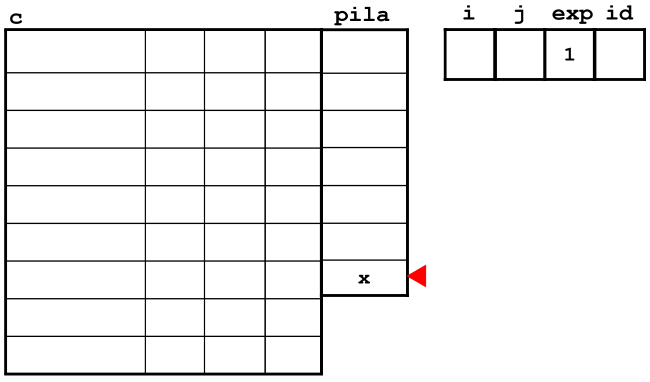
S_1={
 pop exp;
 pop id;
 generate_tuple(
       “(:=,”,exp,
       ”, ,” ,id ”)”;
 i:= next_quadruple();
 push id;
 }



Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10, 2
  z:=1;
  y:=2
end
```

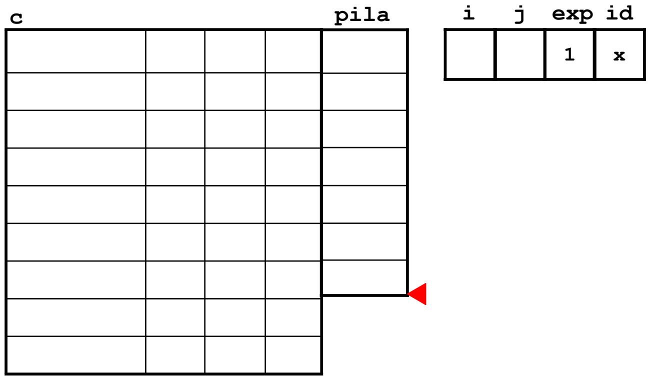


```
S1={
 pop exp;
 pop id;
 generate_tuple(
       "(:=,",exp,
       ", ," ,id ")";
 i:= next_quadruple();
 push id;
 }
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10, 2
  z:=1;
  y:=2
end
```

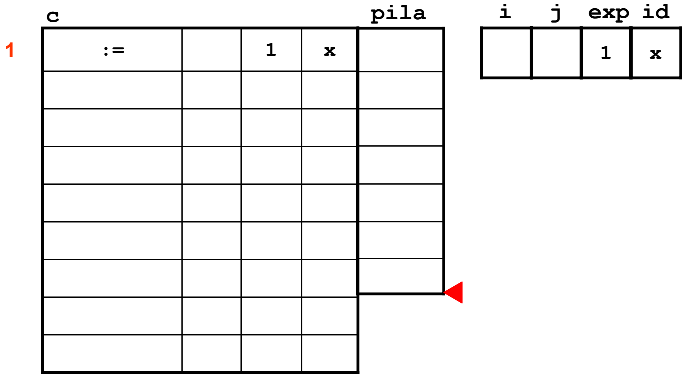


```
S1={
 pop exp;
 pop id;
 generate_tuple(
       "(:=,",exp,
       ", ," ,id ")";
 i:= next_quadruple();
 push id;
 }
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10, 2
  z:=1;
  y:=2
end
```

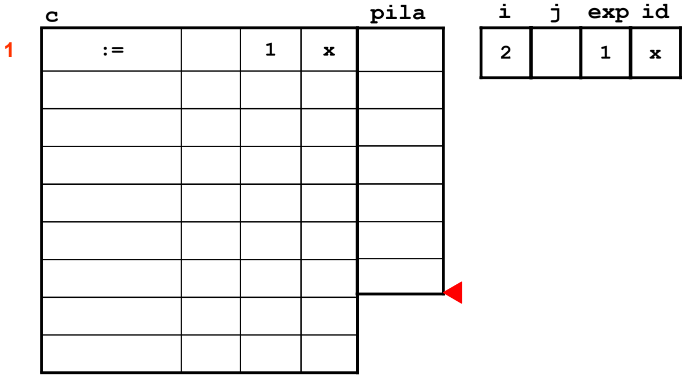


```
S1={
 pop exp;
 pop id;
 generate_tuple(
       "(:=,",exp,
       ", ," ,id ")";
 i:= next_quadruple();
 push id;
 }
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10, 2
  z:=1;
  y:=2
end
```



```
S1={
 pop exp;
 pop id;
 generate_tuple(
       "(:=,",exp,
       ", ," ,id ")";
 i:= next_quadruple();
 push id;
 }
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10, 2
   z:=1;
   y:=2
end
```

	c			
1	:=		1	x

pila
x

i	j	exp	id
2		1	x

```
S1={
 pop exp;
 pop id;
 generate_tuple(
      "(:=,",exp,
      ", ," ,id ")";
 i:= next_quadruple();
 push id;
 }
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10, 2
   z:=1;
   y:=2
end
```

	c			
1	:=		1	x

pila
10
x

i	j	exp	id
2		1	x

```
S1={
 pop exp;
 pop id;
 generate_tuple(
      "(:=,",exp,
      ", ," ,id ")";
 i:= next_quadruple();
 push id;
 }
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10{S2}, 2
   z:=1;
   y:=2
end
```

	c			
1	:=		1	x

pila
10
x

i	j	exp	id
2		1	x

```
S2={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    "(jump_if_greater,," ,id, ",",  exp
")");
 generate_tuple("(jump_to,,,)");
 push id;
}
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10{S2}, 2
   z:=1;
   y:=2
end
```

	c			
1	:=		1	x

pila
x

i	j	exp	id
2		10	x

```
S2={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    "(jump_if_greater,," ,id, ",",  exp
")");
 generate_tuple("(jump_to,,,)");
 push id;
}
```

Quadruples: loops

Examples

```
do x {$S_0$}:=1{$S_1$}, 10{$S_2$}, 2
  z:=1;
  y:=2
end
```

	c			
1	:=		1	x

pila

i	j	exp	id
2		10	x

```
$S_2$={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    “(jump_if_greater,,” ,id, “,”, exp
“)”);
 generate_tuple(“(jump_to,,,)”);
 push id;
}
```

Quadruples: loops

Examples

```
do x {$S_0$}:=1{$S_1$}, 10{$S_2$}, 2
  z:=1;
  y:=2
end
```

	c			
1	:=		1	x

pila

i	j	exp	id
2	2	10	x

```
$S_2$={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    “(jump_if_greater,,” ,id, “,”, exp
“)”);
 generate_tuple(“(jump_to,,,)”);
 push id;
}
```

Quadruples: loops

Examples

```
do x {$S_0$}:=1{$S_1$}, 10{$S_2$}, 2
  z:=1;
  y:=2
end
```

	c			
1	:=		1	x
2	jump_>		x	10

pila

i	j	exp	id
2	2	10	x

```
$S_2$={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    “(jump_if_greater,,” ,id, “,”, exp
“)”);
 generate_tuple(“(jump_to,,,)”);
 push id;
}
```

Quadruples: loops

Examples

```
do x {$S_0$}:=1{$S_1$}, 10{$S_2$}, 2
  z:=1;
  y:=2
end
```

	c			
1	:=		1	x
2	jump_>		x	10
3	jump_to			

pila

i	j	exp	id
2	2	10	x

```
$S_2$={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    “(jump_if_greater,,” ,id, “,”, exp
“)”);
 generate_tuple(“(jump_to,,,)”);
 push id;
}
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10{S2}, 2
  z:=1;
  y:=2
end
```

c

1	:=		1	x
2	jump_>		x	10
3	jump_to			

pila
x

i	j	exp	id
2	2	10	x

```
S2={
 pop exp;
 pop id;
 j:=next_quadruple();
 generate_tuple(
    “(jump_if_greater,,” ,id, “,”,  exp
“)”);
 generate_tuple(“(jump_to,,,)”);
 push id;
}
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10{S2}, 2
  z:=1;
  y:=2
end
```

c

1	:=		1	x
2	jump_>		x	10
3	jump_to			

pila
2
x

i	j	exp	id
2	2	10	x

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10{S2}, 2{S3}
  z:=1;
  y:=2
end
```

c

1	:=		1	x
2	jump_>		x	10
3	jump_to			

pila
2
x

i	j	exp	id
2	2	10	x

```
S3={
  pop exp;
  pop id;
  generate_tuple(
   “(+,”,id “,”,
     ex “,”,id “)”);
  generate_tuple(
   “(jump_to,” , i, “,,)” );
  c[j+1][2]:=next_quadruple();
}
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10{S2}, 2{S3}
  z:=1;
  y:=2
end
```

c

1	:=		1	x
2	jump_>		x	10
3	jump_to			

pila
x

i	j	exp	id
2	2	2	x

```
S3={
  pop exp;
  pop id;
  generate_tuple(
   “(+,”,id “,”,
     ex “,”,id “)”);
  generate_tuple(
   “(jump_to,” , i, “,,)” );
  c[j+1][2]:=next_quadruple();
}
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10{S2}, 2{S3}
  z:=1;
  y:=2
end
```

c

1	:=		1	x
2	jump_>		x	10
3	jump_to			

pila

i	j	exp	id
2	2	2	x

```
S3={
  pop exp;
  pop id;
  generate_tuple(
   “(+,”,id “,”,
     ex “,”,id “)”);
  generate_tuple(
   “(jump_to,” , i, “,,)” );
  c[j+1][2]:=next_quadruple();
}
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10{S2}, 2{S3}
  z:=1;
  y:=2
end
```

c

1	:=		1	x
2	jump_>		x	10
3	jump_to			
4	+	x	2	x

pila

i	j	exp	id
2	2	2	x

```
S3={
  pop exp;
  pop id;
  generate_tuple(
   “(+,”,id “,”,
     ex “,”,id “)”);
  generate_tuple(
   “(jump_to,” , i, “,,)” );
  c[j+1][2]:=next_quadruple();
}
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10{S2}, 2{S3}
  z:=1;
  y:=2
end
```

c

1	:=		1	x
2	jump_>		x	10
3	jump_to			
4	+	x	2	x
5	jump_to	2		

pila

i	j	exp	id
2	2	2	x

```
S3={
  pop exp;
  pop id;
  generate_tuple(
   “(+,”,id “,”,
     ex “,”,id “)”);
  generate_tuple(
   “(jump_to,” , i, “,,)” );
  c[j+1][2]:=next_quadruple();
}
```

Quadruples: loops

Examples

```
do x {S0}:=1{S1}, 10{S2}, 2{S3}
  z:=1;
  y:=2
end
```

c

1	:=		1	x
2	jump_>		x	10
3	jump_to	6		
4	+	x	2	x
5	jump_to	2		

pila

i	j	exp	id
2	2	2	x

```
S3={
  pop exp;
  pop id;
  generate_tuple(
   “(+,”,id “,”,
     ex “,”,id “)”);
  generate_tuple(
   “(jump_to,” , i, “,,)” );
  c[j+1][2]:=next_quadruple();
}
```

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2}, 2{S_3}
  z:=1;
  y:=2
end

	c				pila
1	:=		1	x	
2	jump_>		x	10	
3	jump_to	6			
4	+	x	2	x	
5	jump_to	2			
6	:=	1		z	

i	j	exp	id
2	2	2	x

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2}, 2{S_3}
  z:=1;
  y:=2
end

	c				pila
1	:=		1	x	
2	jump_>		x	10	
3	jump_to	6			
4	+	x	2	x	
5	jump_to	2			
6	:=	1		z	
7	:=	2		y	

i	j	exp	id
2	2	2	x

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2}, 2{S_3}
  z:=1;
  y:=2
end {S_5}

	c				pila
1	:=		1	x	
2	jump_>		x	10	
3	jump_to	6			
4	+	x	2	x	
5	jump_to	2			
6	:=	1		z	
7	:=	2		y	

i	j	exp	id
2	2	2	x

S_5={
  generate_tuple(“(jump_to,” ,j+2, “,,)”);
  c[j][2]:=next_quadruple();
}

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2}, 2{S_3}
  z:=1;
  y:=2
end {S_5}

	c				pila
1	:=		1	x	
2	jump_>		x	10	
3	jump_to	6			
4	+	x	2	x	
5	jump_to	2			
6	:=	1		z	
7	:=	2		y	
8	jump_to	4			

i	j	exp	id
2	2	2	x

S_5={
  generate_tuple(“(jump_to,” ,j+2, “,,)”);
  c[j][2]:=next_quadruple();
}

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2}, 2{S_3}
  z:=1;
  y:=2
end {S_5}

	c				pila
1	:=		1	x	
2	jump_>	9	x	10	
3	jump_to	6			
4	+	x	2	x	
5	jump_to	2			
6	:=	1		z	
7	:=	2		y	
8	jump_to	4			

i	j	exp	id
2	2	2	x

S_5={
  generate_tuple(“(jump_to,” ,j+2, “,,)”);
  c[j][2]:=next_quadruple();
}

Quadruples: loops

Examples

do x {S_0}:=1{S_1}, 10{S_2}, 2{S_3}
  z:=1;
  y:=2
end {S_5}

	c				pila
1	:=		1	x	
2	jump_>	9	x	10	
3	jump_to	6			
4	+	x	2	x	
5	jump_to	2			
6	:=	1		z	
7	:=	2		y	
8	jump_to	4			

i	j	exp	id
2	2	2	x

Bibliography

Bibliography

[Alf] “*Teoría de Autómatas y lenguajes formales*” M. Alfonseca y otros

[Hop] *“Introducción a la teoría de autómatas, lenguajes y computación*” Hopcroft, J.; Motwani, R.; Ullman, J.