Lesson 6 part a: generation of intermediate code

Intermediate Representations

INDEX

- Introduction
- Postfix (suffix) notation
- Tuples

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
Let us consider some introductory examples to intermediate representations, focusing on arithmetic expressions.

Consider the following ASPLE expression:

\[ a \times (9 + d) \]

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

\[ a * (9 + d) \iff a9d+* \]

\[ \begin{array}{c}
    \text{d} \\
    \text{9} \\
    \text{a}
\end{array} \]

\[ \begin{array}{c}
    \text{9+d} \\
    \text{9+d} \\
    \text{a}
\end{array} \]
Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

```
a \ast ( \, 9 + \, d \, ) \iff a9d+*
```

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

\[ a := a \times (9 + d) \iff a = a \times 9 + a \times d \]
Intermediate Representations

Introductory examples: extension to other types of expressions

\[ a := a \ast (9 + d) \Leftrightarrow a \ast (9 + d) \]

\[ a := a \ast (9 + d) \Leftrightarrow a \ast (9 + d) \]

\[ a := a \ast (9 + d) \Leftrightarrow a \ast (9 + d) \]
Intermediate Representations

Introductory examples: extension to other types of expressions

\[ a := a \times (9 + d) \]

Intermediate Representations

Introductory examples: extension to other types of expressions

\[ a := a \times (9 + d) \]

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...
  \[ a := a \times (9 + d) \]

- could be represented with the following sequence of steps (quadruples):
  \[ (+, 9, d, \text{output}\_\text{step1}) \]
  \[ (*, a, \text{output}\_\text{step2}, a) \]

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.
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.

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`
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.

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.

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:

\[
\begin{align*}
E & \rightarrow E + T \\
E & \rightarrow E - T \\
E & \rightarrow T \\
T & \rightarrow T * F \\
T & \rightarrow T / F \\
T & \rightarrow F \\
F & \rightarrow i \\
F & \rightarrow (E) \\
F & \rightarrow -F
\end{align*}
\]

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

\[
\begin{align*}
E & \rightarrow E + T \text{(push +)} \\
E & \rightarrow E - T \text{(push -)} \\
E & \rightarrow T \\
T & \rightarrow T * F \text{(push *)} \\
T & \rightarrow T / F \text{(push /)} \\
T & \rightarrow F \\
F & \rightarrow i \text{(push i)} \\
F & \rightarrow (E) \\
F & \rightarrow -F \text{(push _)} (*)
\end{align*}
\]

(*) Although the same symbol represents the subtraction and the negation, they are different operations, so we use different symbols for the abstract stack machine.
• Se puede comprobar que las siguientes acciones semánticas:

- $E \rightarrow E + T$
- $E \rightarrow E - T$
- $E \rightarrow T$
- $T \rightarrow T \cdot F$
- $T \rightarrow T / F$
- $T \rightarrow F$
- $F \rightarrow i$
- $F \rightarrow (E)$
- $F \rightarrow -F$

Generación de la representación sufija asociada al programa:

- push $a$
- push $b$
- push $c$
- push $a$
El texto en el documento se puede leer así: 

**Suffix notation**

- Se pueden comprobar que las siguientes acciones semánticas:
  - $E \rightarrow E+T$
  - $E \rightarrow E-T$
  - $E \rightarrow T$
  - $T \rightarrow T*F$
  - $T \rightarrow T/F$
  - $T \rightarrow F$
  - $F \rightarrow i$
  - $F \rightarrow (E)$
  - $F \rightarrow -F$

**Generación de la representación sufija asociada al programa**

-推 $a$
-推 $b$
-推 $c$
-推 $a$
-推 /

**Suffix notation**

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

  $a$ $b$ $c$ $a$ / + *
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.

In our example:

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

In our example:

```
push dword [a]
push dword [b]
push dword [c]
pop  eax
pop  ebx
div  ebx, eax, eax
push dword eax
pop  eax
pop  ebx
mull ebx, eax, eax
push eax
```
**Suffix notation**

Code generation, associated to the intermediate representation

- In our example:

```
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
idiv ebx, eax
push eax
add  ebx, eax
push eax
pop  eax
pop  ebx
mul  ebx, eax
push eax
```
The following "attributes grammar" generates the sequence of PC Assembly instructions:

\[\text{Operand} \rightarrow \text{i \{printf("push dword \%s\n",i.name)(1)\}}\]

\[\text{Operand} \rightarrow \text{cte \{printf("push dword \%s\n",cte.name)(1)\}}\]

\[\text{Operand} \rightarrow \text{Operand} \text{ Operand} \text{ Dyadic Operator \{if ("the operation is valid, (types, etc...")}{\}
\text{printf("pop eax\n"),}
\text{printf("pop ebx\n"),}
\text{printf("%s ebx eax\n",<Dyadic Operator>.symbol)(2)}
\text{printf("push eax\n")\}}\]

\[\text{else "ERROR"}\]

\[\text{Operand} \rightarrow \text{Operand} \text{ Monadic Operator \{if ("the operation is valid, (types, etc...")}{\}
\text{printf("pop eax\n"),}
\text{printf("%s eax\n",<Monadic Operator>.symbol)}
\text{printf("push eax\n")}]}\]

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.

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

\[\text{identifier := expression}\]

Consider this statement for assigning a value to a.

\[a := a*(b+c/a)\]

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: unconditional jump

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

  \texttt{jump\_to <Label>}

• In our example,

  \texttt{push dword Label}
  \texttt{pop eax}
  \texttt{jmp near eax}

• In our example,

  \texttt{jump\_to Label}

Suffix notation

Code generation, associated to the intermediate representation: conditional jump

• Remember the syntax of the conditional jumps:

  \texttt{"condition"}
  \texttt{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:
• In our example,

```
push dword Label
pop eax
jz near eax
```

(*)

jump_if_false Label

---

• 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:
In our example, associated to the intermediate representation:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>L1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>L1</td>
<td>jump_if_false</td>
<td>inst1</td>
<td>L2</td>
<td>jump_to</td>
<td>L1:</td>
</tr>
</tbody>
</table>

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

---

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>L1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>L1</td>
<td>jump_if_false</td>
<td>inst1</td>
<td>L2</td>
<td>jump_to</td>
<td>L1:</td>
</tr>
</tbody>
</table>

"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

```
<table>
<thead>
<tr>
<th>p</th>
<th>L1</th>
<th>jump_if_false</th>
<th>inst1</th>
<th>L2</th>
<th>jump_to</th>
<th>L1:</th>
<th>inst2</th>
<th>L2:</th>
</tr>
</thead>
</table>
```

```
“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:
```

```
L2:
```

```
jump_if_false
```

```
L1: inst2
jump_to
L2:
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

---

**In our example**

```
<table>
<thead>
<tr>
<th>p</th>
<th>L1</th>
<th>jump_if_false</th>
<th>inst1</th>
<th>L2</th>
<th>jump_to</th>
<th>L1:</th>
<th>inst2</th>
<th>L2:</th>
</tr>
</thead>
</table>
```

```
“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:
```

```
L2:
```

```
jump_if_false
```

```
L1: inst2
jump_to
L2:
```

```
```

```
```

```
```

```
```

---

**In our example**

```
<table>
<thead>
<tr>
<th>p</th>
<th>L1</th>
<th>jump_if_false</th>
<th>inst1</th>
<th>L2</th>
<th>jump_to</th>
<th>L1:</th>
<th>inst2</th>
<th>L2:</th>
</tr>
</thead>
</table>
```

```
“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:
```

```
L2:
```

```
jump_if_false
```

```
L1: inst2
jump_to
L2:
```

```
```

```
```

```
```

---

**In our example**

```
<table>
<thead>
<tr>
<th>p</th>
<th>L1</th>
<th>jump_if_false</th>
<th>inst1</th>
<th>L2</th>
<th>jump_to</th>
<th>L1:</th>
<th>inst2</th>
<th>L2:</th>
</tr>
</thead>
</table>
```

```
“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:
```

```
L2:
```

```
jump_if_false
```

```
L1: inst2
jump_to
L2:
```

```
```

```
```

```
```

---

**In our example**

```
<table>
<thead>
<tr>
<th>p</th>
<th>L1</th>
<th>jump_if_false</th>
<th>inst1</th>
<th>L2</th>
<th>jump_to</th>
<th>L1:</th>
<th>inst2</th>
<th>L2:</th>
</tr>
</thead>
</table>
```

```
“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:
```

```
L2:
```

```
jump_if_false
```

```
L1: inst2
jump_to
L2:
```

```
```

```
```

```
```

---
• Introduction

• Postfix (suffix) notation

• Tuples

---

**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:

**Example**

Consider the following attributes grammar:

\[
G_3 = \langle E.v, T.v, F.v \rangle, \{i.name, +.simb, -.simb, *.simb, /.simb, (,) \}
\]

\[
\begin{align*}
E \rightarrow & E + T \quad \{ \text{If valid +} \ add("+.symbol,E.v,T.v,id=new\_symbol()"), \\
\quad & E.v=id \} \\
E \rightarrow & E - T \quad \{ \text{If valid -} \ add("-.symbol,E.v,T.v,id=new\_symbol()"), \\
\quad & E.v=id \} \\
E \rightarrow & T \quad \{ E.v=T.v \} \\
T \rightarrow & T * F \quad \{ \text{If valid *} \ add("*.symbol,T.v,F.v,id=new\_symbol()"), \\
\quad & T.v=id \} \\
T \rightarrow & T / F \quad \{ \text{If valid /} \ add("/.symbol,T.v,T.v,id=new\_symbol()"), \\
\quad & T.v=id \} \\
T \rightarrow & F \quad \{ T.v=F.v \} \\
F \rightarrow & i \quad \{ F.v=i.name \} \\
F \rightarrow & (E) \quad \{ F.v=E.v \} \\
F \rightarrow & F " \quad \{ \text{If valid _} \ add("_.symbol,F.v,id=new\_symbol()"), \\
\quad & F.v=id \} \\
E & >
\end{align*}
\]
Generation of tuples in bottom-up analysis

Example

- The quadruples are generated correctly:
  
  \[ F.v=a \]

Generation of tuples in bottom-up analysis

Example

- The quadruples are generated correctly:
  
  \[ F.v=a \]
  \[ T.v=a \]

Generation of tuples in bottom-up analysis

Example

- The quadruples are generated correctly:
  
  \[ F.v=a \]
  \[ T.v=a \]
  \[ F.v=b \]

Generation of tuples in bottom-up analysis

Example

- The quadruples are generated correctly:
  
  \[ T.v=a \]
  \[ F.v=a \]
  \[ F.v=b \]
Generation of tuples in bottom-up analysis

Example

F.v=a
T.v=a
F.v=b
F.v=b
F.v=b
F.v=c
F.v=c
E.v=t₁

(+, b, c, t₁)
Generation of tuples in bottom-up analysis

Example

\[(+,b,c,t_1)\]

\[F.v=b\]

\[F.v=b\]

\[F.v=b\]

\[F.v=c\]

\[E.v=t_1\]

\[F.v=t_1\]

\[T.v=t_1\]

Generation of tuples in bottom-up analysis

Example

\[(+,b,c,t_1)\]

\[F.v=b\]

\[F.v=b\]

\[F.v=t_1\]

\[F.v=c\]

\[E.v=t_1\]

\[T.v=t_2\]

\[F.v=t_1\]

\[T.v=t_2\]

\[E.v=t_2\]

Generation of tuples in bottom-up analysis

Example

\[(+,b,c,t_1)\]

\[F.v=b\]

\[F.v=b\]

\[F.v=t_1\]

\[F.v=c\]

\[E.v=t_1\]

\[T.v=t_1\]

\[F.v=c\]

\[T.v=t_2\]

\[E.v=t_2\]

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:

\[G_3 = \langle E, T, F \rangle, \{+, -, *, /, (,)\}\]

\[E \rightarrow T + E \mid T - E \mid T\]

\[T \rightarrow F * T \mid F / T \mid F\]

\[F \rightarrow i \mid (E)\]

\[E>\]
It had the following LL(1) equivalent grammar:

\[ G_3 = \langle E, T, F, i, +, -, *, /, (,) \rangle \]

\[ E \rightarrow \text{iV} \mid (ECV) \]

\[ V \rightarrow +TX \mid -E \mid \lambda \]

\[ X \rightarrow +E \mid -E \mid \lambda \]

\[ T \rightarrow \text{iU} \mid (ECU) \]

\[ U \rightarrow \text{*T} \mid /T \mid \lambda \]

\[ F \rightarrow \text{i} \mid (EC) \]

\[ C \rightarrow \lambda \]

Example

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

```c
int E(char * string, int i) {
    // Propagate previous errors
    if (i < 0) return i;
    switch (string[i]) {
        case 'i':
            i++;
            i=E(string, i);
            break;
        case '+':
            i++;
            i=E(string, i);
            i=V(string, i);
            break;
        default: return -1; /* no \lambda */
    }
    return i;
}
```

```c
int V(char * string, int i) {
    // Propagate previous errors
    if (i < 0) return i;
    switch (string[i]) {
        case '+':
        case '-':
        i++;
        i=T(string, i);
        break;
        case '(': i++;
        i=E(string, i);
        break; /* \lambda */
    }
    return i;
}
```

```c
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; /* \lambda */
    }
    return i;
}
```

```c
int T(char * string, int i) {
    // Propagate previous errors
    if (i < 0) return i;
    switch (string[i]) {
        case '+':
        case '-':
        i++;
        i=E(string, i);
        break; /* \lambda */
    }
    return i;
}
```

```c
int U(char * string, int i) {
    // Propagate previous errors
    if (i < 0) return i;
    switch (string[i]) {
        case '+':
        case '-':
        i++;
        i=E(string, i);
        i=C(string, i);
        break;
        default: return -2; /* no \lambda */
    }
    return i;
}
```

```c
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; /* \lambda */
        default: return -3; /* no \lambda */
    }
    return i;
}
```
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;
}

Example

• The string is analysed with the following function call
  \texttt{axiom(a(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:

\begin{verbatim}
0[1]: E( "I + I * I", 0)
1[1]: E( "I + I * I", 1)
2[1]: E( "I + I * I", 2)
3[1]: E( "I + I * I", 3)
4[1]: E( "I + I * I", 4)
5[1]: E( "I + I * I", 5)
5[1]: returned 5
4[1]: returned 5
4[1]: X( "I + I * I", -2)  
4[1]: returned -2
3[1]: returned -2
2[1]: returned -2
1[1]: returned -2
0[1]: returned -2
-2
\end{verbatim}

Example

• Incorrect string:

\begin{verbatim}
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
\end{verbatim}

Example

• The tuples can be generated in the middle of the code (in \textcolor{red}{red}) as follows:

\begin{verbatim}
int E(char * string, int i)
{
    // Propagate previous errors
    if (i < 0) return i;
    switch (string[i]) {
    case 'i':
        push(i.name);
        i++;
        break;
    case '+':
        case '-':
            j=i;
            i=E(string, i);
            quad(string[j].pop(), pop(), gen(Ti));
            push(Ti);
            break;
        case '*':
        case '/':
            j=i;
            i=E(string, i);
            i=C(string, i);
            i=W(string, i);
            i=X(string, i);
            quad(string[j].pop(), pop(), gen(Ti));
            push(Ti);
            break; /*λ*/
    }
    return i;
}
\end{verbatim}

Example

\begin{verbatim}
int V(char * string, int i)
{
    // Propagate previous errors
    if (i < 0) return i;
    switch (string[i]) {
    case 'i':
        push(i.name);
        i++;
        i=E(string, i);
        i=C(string, i);
        i=W(string, i);
        break;
    default: return -1; /*no λ*/
    }
    return i;
}
\end{verbatim}
Generation of tuples in selective top-down analysis

Example

```c
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=E(string, i);
            quad(string[j], pop(), pop(), gen(Ti));
            push(Ti);
            break; /* λ */
        default: return i;
    }
    return i;
}
```

```c
int T(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=E(string, i);
            quad(string[j], pop(), pop(), gen(Ti));
            push(Ti);
            break; /* λ */
        default: return i;
    }
    return i;
}
```

```c
int U(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=E(string, i);
            quad(string[j], pop(), pop(), gen(Ti));
            push(Ti);
            break; /* λ */
        default: return i;
    }
    return i;
}
```

```c
int C(char * string, int i)
{
    // Propagate previous errors
    if (i < 0) return i;
    switch (string[i]) {
        case ')':
            i++;
            return i;
        default: return -4; /* no λ */
    }
    return i;
}
```

Example

- The string \( x \) is analysed with the following call:

  ```c
  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:
Generation of tuples in selective top-down analysis

Example

\text{axiom( }x, 0;\text{ )}

\begin{verbatim}
113
E("1 + i * i", 0)
0[1]: E( "1 + I * I", 0 )
1[1]: Vl "1 + I * I", 1
2[1]: E( "I + I * I", 2 )
3[1]: Vl "I + I * I", 3
4[1]: Tl "I + I * I", 4
5[1]: U( "I + I * I", 5 )
5[1]: returned 5
4[1]: returned 5
4[1]: Xl "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
\end{verbatim}

Example

\text{axiom( }x, 0;\text{ )}

\begin{verbatim}
116
E("1 + i * i", 0)
0[1]: E( "1 + I * I", 0 )
1[1]: Vl "1 + I * I", 1
2[1]: E( "I + I * I", 2 )
3[1]: Vl "I + I * I", 3
4[1]: Tl "I + I * I", 4
5[1]: U( "I + I * I", 5 )
5[1]: returned 5
4[1]: returned 5
4[1]: Xl "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
\end{verbatim}
Example

\begin{align*}
\text{axiom(} x, 0) ;
\end{align*}
Generation of tuples in selective top-down analysis

Example

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

Example

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

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```

Example

```plaintext
(*,i,i,t1) t1
```
Generation of tuples in selective top-down analysis

Example

\[
\text{axiom}(x, 0);
\]

\[
\begin{align*}
E & ("I + I * I", 0) \\
0[1] & : E("I + I * I", 0, 1) \\
1[1] & : V("I + I * I", 1) \\
2[1] & : E("I + I * I", 2) \\
3[1] & : V("I + I * I", 3) \\
4[1] & : T("I + I * I", 4) \\
5[1] & : U("I + I * I", 5) \\
6[1] & : \text{returned } 5 \\
4[1] & : \text{returned } 5 \\
4[1] & : X("I + I * I", 5) \\
4[1] & : \text{returned } 5 \\
3[1] & : \text{returned } 5 \\
2[1] & : \text{returned } 5 \\
1[1] & : \text{returned } 5 \\
0[1] & : \text{returned } 5 \\
5 & : \text{returned } 5
\end{align*}
\]

Tuples

General concepts: available operations

- Usually, we can find:
  - Operations to assign an expression to a variable:
    - Examples:
      \[
      \begin{align*}
      x & := y \text{ <Binary_operator> } z \\
      x & := \text{ <Monadic_operator> y} \\
      x & := y
      \end{align*}
      \]
    - 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.

- Unconditional jumps:
  \[
  \text{jump_to } \text{<Label>}
  \]

- Conditional jumps:
  \[
  \text{<operation>}
  \text{if } \text{<condition> } \text{jump_to } \text{<Label>}
  \]
  - 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:
    \[
    \text{if } \text{<logic_operation> then } \text{<statements> else } \text{<statementsElse>}
    \]
    - In this case, if the operation is correct, then \(<\text{statements}>\) will execute. Otherwise, \(<\text{statementsElse}>\) 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:
    - \((\text{Operator}, \text{Operand}_1, \text{Operand}_2, \text{Result})\)
  • **Triplets**, with the structure
    - \((\text{Operator}, \text{Operand}_1, \text{Operand}_2)\)
    • 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.

Quadruples

Semantics of conditional flow statements

• With respect to the conditional flow,
  • let us see the following attributes grammar:

\[
G_3 =< \{\text{Instr}, \text{Expr}\},
\{\text{if }\}
\{\text{Instr}\rightarrow\text{if }\text{Expr}\{S_2\}\text{then }\text{Instr}\{S_1\}
\text{Instr}\rightarrow\text{if }\text{Expr}\{S_2\}\text{then }\text{Instr}\{S_3\}\text{Instr}\{S_1\}
\}
,\text{Instr}\>
\]

Tuples

General concepts

• We shall use the following set of quadruples:
  • \((\text{jump}_{\text{if}}\text{condition}, \text{jump_address}, \text{Operand})\)
    • for conditional jumps to \text{jump_address} according to the value of \text{Operand}
  • \((\text{jump}_{\text{to}}, \text{jump_address}, \, )\)
    • for unconditional jumps.
  • \((\text{:=}, \text{expression}, \text{target})\)
    • For assigning the value of \text{expression} to the variable \text{target}
  • \((\text{Dyadic}_{\text{operator}}, \text{Operand}_1, \text{Operand}_2, \text{target})\)
    • for the typical arithmetic and logic operations.
  • \((\text{Monadic}_{\text{operator}}, \text{Operand}, \text{target})\)
    • For the typical monadic operations

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 \text{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 \text{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 \text{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 \text{id_quadruple} to store identifiers of quadruples.
Quadruples

Semantics of conditional flow statements

\[ S_1 = \{ \text{c}\{\text{top}\}[2]=\text{next}_\text{quadruple}(); \}
\text{pop;} \}
\]

\[ S_2 = \{ \text{pop} \text{id}; /* id contains the result of <Expr> */ \]
\[ \text{id}_\text{quadruple} = \text{generate}_\text{tuple}("\text{jump}_\text{if}_\text{false},,,,"\text{id}""); \]
\[ \text{push id}_\text{quadruple}; \}
\]

\[ S_3 = \{ \text{id}_\text{quadruple} = \text{generate}_\text{tuple}("\text{jump}_\text{to},,,") ; \]
\[ \text{c}\{\text{top}\}[2]=\text{next}_\text{quadruple}(); \]
\[ \text{pop}; \]
\[ \text{push id}_\text{quadruple}; \}
\]

\[ G_3 = \{ \langle \text{Instr}, \langle \text{Expr} \rangle \rangle , \{ \text{if} \} \}
\{ \text{<Instr>→if}<\text{Expr}>[S_2]\text{then}<\text{Instr}>[S_1] \}
\{ \text{<Instr>→if}<\text{Expr}>[S_2]\text{then}<\text{Instr}'>[S_3]<\text{Instr}>[S_1] \}, \}
\text{<Instr> >}
\]

Quadruples: loops

Examples

- Apply the attributes grammar to the next program:

\[
\text{if} \ (a + b) < (c * d) \]
\[ \text{then} \]
\[ a := a / b \]
\[ \text{else} \]
\[ a := a * b \]
\[
\text{if} \ (a < b) \]
\[ \text{then} \]
\[ x := 1 \]
\[ \text{else} \]
\[ x := 2 \]
\[
\text{if } E_1 \]
\[ \text{then} \]
\[ I_1 \]

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:

\[ G_3 = \{ \langle \text{Instr} \rangle , \}
\{ \text{id, goto, ":"} \}
\{ \}
\{ \text{<Instr>→id : <Instr> \{ \text{declare_label(id)} ; \}} \}
\text{<Instr>→goto id \{ \text{goto_identifier(id)} ; \}}, \}
\text{<Instr> >}
\]

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 \text{TS}
  - A data type \text{value} for the values stored in the symbols table.
- Functions to access the symbols table:
  - \text{insert(identifier id, value value, table TS)};
  - \text{change(identifier id, value new_value, table TS)};
  - \text{value search(identifier id, table TS)};
- We shall assume that the data type \text{value} contains:
  - Identifier type (\text{type}): \text{label}
  - A flag to indicate whether it has been declared or not.
  - The label of the next quadruple (\text{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 (`nº_quadruple`)
- A function `generate_tuple(tuple)` that inserts the tuple provided as argument in the next position in the matrix `c`.
- A function:
  ```c
  define_label(identifier label, nº_quadruple number)
  ```
  - that associates a label to a certain quadruple number.
- A function
  ```c
  error();
  ```
  - that starts the treatment of an error condition.

---

**Examples**

- Apply the attributes grammar to the next case:

  ```c
  goto L1
  ...goto L2
  ...
  L1:···
  ...
  L2:···
  goto L1
  ...
  goto L2
  ...
  ```

  ```c
  insert(L1,("undeclared", 1))
  insert(L2,("undeclared", n1))
  ...L1:···
  1 (jump_to, n2, )
  ...
  ...L2:···
  ...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:

\[
G_3 = \langle \{ \text{Instr}, \text{Expr}, \text{CD1}, \text{Linstr} \}, \\
\{ \text{do}, \text{id}, :=, \text{,} \text{,} \text{,} \text{,} \}\, \rangle
\]

- Sequences of \text{Instr} separated by \text{;}  \\
- \text{Arithmetic expressions}

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 \text{push} and \text{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 \text{c}.
- Auxiliary variables \text{i} and \text{j} with the same type as the labels of the quadruples.
- An auxiliary variable \text{exp} to store the value of arithmetic expressions.
- An auxiliary variable \text{id} to store the names of the identifiers.
- A function \text{quadruple_label next_quadruple();} that returns a new label to identify a quadruple, corresponding to the first empty position in matrix \text{c}.
- A function \text{generate_tuple(tup);} to insert, in the next empty position of the matrix \text{c}, the tuple provided as argument.

Quadruples

Semantics of iterative control flow.

\[
S_0 = \{ \text{push id;} \} \\
S_1 = \{ \text{pop exp; pop id;} \\
\text{generate_tuple(":=","exp","","id");} \\
i := \text{next_quadruple();} \\
\text{push id;}
\}
\]

\[
S_2 = \{ \text{pop exp; pop id;} \\
j := \text{next_quadruple();} \\
\text{generate_tuple("jump_if_greater","id","","exp");} \\
\text{generate_tuple("jump_to",""});} \\
\text{push id;}
\}
\]

\[
S_3 = \{ \text{pop exp;} \\
\text{pop id;} \\
\text{generate_tuple("(+,id","","","id")");} \\
\text{generate_tuple("(jump_to,"",i,"",")");} \\
c[j+1][2] := \text{next_quadruple();}
\}
\]

Quadruples

Semantics of iterative control flow.

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

\[
S_5 = \{ \text{generate_tuple("(jump_to,"",j2,"",")");} \\
c[j][2] := \text{next_quadruple();}
\}
\]
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.

\[
\text{do } x := 1, 10 \\
\quad z := 1; \\
\quad y := 2 \\
\text{end}
\]

\[
\text{do } x := 1, 10, 2 \\
\quad z := 1; \\
\quad y := 2 \\
\text{end}
\]
quadruples: loops

examples

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

```
S₀=
  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;
}

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

S_2={
    pop exp;
pop id;
j:=next_quadruple();
generate_tuple("(jump_if_greater,,id,,", exp ");
generate_tuple("(jump_to,,)");
push id;
}

Examples

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

S_2={
    pop exp;
pop id;
j:=next_quadruple();
generate_tuple("(jump_if_greater,,id,,", exp ");
generate_tuple("(jump_to,,)");
push id;
}

Examples

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

S_2={
    pop exp;
pop id;
j:=next_quadruple();
generate_tuple("(jump_if_greater,,id,,", exp ");
generate_tuple("(jump_to,,)");
push id;
}
**Quadruples: loops**

Do $x$ \{S_0\}:=1\{S_1\}, 10\{S_2\} \{S_4\}

- $z:=1$;
- $y:=2$

End

Examples

<table>
<thead>
<tr>
<th>c</th>
<th>pila</th>
<th>i</th>
<th>j</th>
<th>exp</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>:=</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>jump &gt;</td>
<td>x</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>jump_to</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$S_4=$

- pop id;
- generate_tuple("(+, id, ",1," ,id ")");
- generate_tuple("(jump_to," ,i, ",")");
- c[j+1][2]:=next_quadruple();

---

**Quadruples: loops**

Do $x$ \{S_0\}:=1\{S_1\}, 10\{S_2\} \{S_4\}

- $z:=1$;
- $y:=2$

End

Examples

<table>
<thead>
<tr>
<th>c</th>
<th>pila</th>
<th>i</th>
<th>j</th>
<th>exp</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>:=</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>jump &gt;</td>
<td>x</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>jump_to</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>jump_to</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$S_4=$

- pop id;
- generate_tuple("(+, id, ",1," ,id ")");
- generate_tuple("(jump_to," ,i, ",")");
- c[j+1][2]:=next_quadruple();

---

**Quadruples: loops**

Do $x$ \{S_0\}:=1\{S_1\}, 10\{S_2\} \{S_4\}

- $z:=1$;
- $y:=2$

End

Examples

<table>
<thead>
<tr>
<th>c</th>
<th>pila</th>
<th>i</th>
<th>j</th>
<th>exp</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>:=</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>jump &gt;</td>
<td>x</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>jump_to</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>x</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>jump_to</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$S_4=$

- pop id;
- generate_tuple("(+, id, ",1," ,id ")");
- generate_tuple("(jump_to," ,i, ",")");
- c[j+1][2]:=next_quadruple();

---

**Quadruples: loops**

Do $x$ \{S_0\}:=1\{S_1\}, 10\{S_2\} \{S_4\}

- $z:=1$;
- $y:=2$

End

Examples

<table>
<thead>
<tr>
<th>c</th>
<th>pila</th>
<th>i</th>
<th>j</th>
<th>exp</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>:=</td>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>jump &gt;</td>
<td>x</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>jump_to</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>x</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>jump_to</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$S_4=$

- 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

S_3={
    generate_tuple( "(jump_to," ,j+2, ",,)");
c[j][2]:=next_quadruple();
}

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

\begin{array}{|c|c|c|c|}
\hline
\text{c} & \text{pila} & \text{i} & \text{j} & \text{exp} & \text{id} \\
\hline
\text{:=} & 1 & x & 2 & 2 & 10 \\
\hline
\text{jump_>} & 9 & x & 10 \\
\hline
\text{jump_to} & 6 \\
\hline
+ & x & 1 & x \\
\hline
\text{jump_to} & 2 \\
\hline
\text{:=} & 1 & x \\
\hline
\text{:=} & 2 & y \\
\hline
\text{jump_to} & 4 \\
\hline
\end{array}

S_5={
\begin{align*}
generate\_tuple\("(jump\_to,\,j+2,\,,\,))"; \\
c[j][2]:=\text{next\_quadruple}(); \\
\end{align*}
}

Examples

\begin{array}{|c|c|c|c|}
\hline
\text{c} & \text{pila} & \text{i} & \text{j} & \text{exp} & \text{id} \\
\hline
\text{:=} & 1 & x & 2 & 2 & 10 \\
\hline
\text{jump_>} & 9 & x & 10 \\
\hline
\text{jump_to} & 6 \\
\hline
+ & x & 1 & x \\
\hline
\text{jump_to} & 2 \\
\hline
\text{:=} & 1 & z \\
\hline
\text{:=} & 2 & y \\
\hline
\text{jump_to} & 4 \\
\hline
\end{array}

Quadruples: loops
Examples

\begin{array}{|c|c|c|c|}
\hline
\text{c} & \text{pila} & \text{i} & \text{j} & \text{exp} & \text{id} \\
\hline
\text{:=} & 1 & x & 2 & 2 & 10 \\
\hline
\text{jump_>} & 9 & x & 10 \\
\hline
\text{jump_to} & 6 \\
\hline
+ & x & 1 & x \\
\hline
\text{jump_to} & 2 \\
\hline
\text{:=} & 1 & z \\
\hline
\text{:=} & 2 & y \\
\hline
\text{jump_to} & 4 \\
\hline
\end{array}

Quadruples: loops
Examples

\begin{array}{|c|c|c|c|}
\hline
\text{c} & \text{pila} & \text{i} & \text{j} & \text{exp} & \text{id} \\
\hline
\text{:=} & 1 & x & 2 & 2 & 10 \\
\hline
\text{jump_>} & 9 & x & 10 \\
\hline
\text{jump_to} & 6 \\
\hline
+ & x & 1 & x \\
\hline
\text{jump_to} & 2 \\
\hline
\text{:=} & 1 & z \\
\hline
\text{:=} & 2 & y \\
\hline
\text{jump_to} & 4 \\
\hline
\end{array}

Quadruples: loops
Examples

\begin{array}{|c|c|c|c|}
\hline
\text{c} & \text{pila} & \text{i} & \text{j} & \text{exp} & \text{id} \\
\hline
\text{:=} & 1 & x & 2 & 2 & 10 \\
\hline
\text{jump_>} & 9 & x & 10 \\
\hline
\text{jump_to} & 6 \\
\hline
+ & x & 1 & x \\
\hline
\text{jump_to} & 2 \\
\hline
\text{:=} & 1 & z \\
\hline
\text{:=} & 2 & y \\
\hline
\text{jump_to} & 4 \\
\hline
\end{array}
Quadruples: loops

Examples

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

S_0={push id;}

d_1=

pila

i  j  exp  id

1  x

Quadruples: loops

Examples

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

S_0={push id;}

d_1=

pila

i  j  exp  id

1  x

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 {S_0} := l{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

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

S_2 =
  pop exp;
pop id;
j := next_quadruple();
generate_tuple("(jump_if_greater,", id, ",", exp "");
generate_tuple("(jump_to,)");
push id;
}
```

```
\[
\begin{array}{c|c|c}
& i & j \\
\hline
1 & := & 1 \\
2 & 10 & x \\
\end{array}
\]
```

```plaintext
S_2 =
  pop exp;
pop id;
j := next_quadruple();
generate_tuple("(jump_if_greater,", id, ",", exp "");
generate_tuple("(jump_to,)");
push id;
}
```

```
\[
\begin{array}{c|c|c}
& i & j \\
\hline
1 & := & 1 \\
2 & 10 & x \\
\end{array}
\]
```

```
\[
\begin{array}{c|c|c}
& i & j \\
\hline
1 & := & 1 \\
2 & 10 & x \\
\end{array}
\]
```
Quadruples: loops

Examples

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

$S_2 =$
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

$S_3 =$
pop exp;
pop id;
generate_tuple("(+,", id ",", ex ",", id ")");
generate_tuple("(jump_to," i ",",
end
ex ",", id ")");
generate_tuple("(jump_to," i ",",
end
ex ",", id ")");
end

c[j+1][2]:=next_quadruple();
}
Quadruples: loops

Examples

\[
\text{do } x \{ S_0 \} := 1 \{ S_1 \}, \ 10 \{ S_2 \}, \ 2 \{ S_3 \} \\
\quad z := 1; \\
\quad y := 2; \\
\text{end}
\]

\[
\begin{array}{|c|c|c|} 
\hline
\text{c} & \text{i} & \text{j} & \text{exp} & \text{id} \\
\hline
1 & := & 1 & x \; & \\
2 & \text{jump}_> & x & 10 & \\
3 & \text{jump}_to & & & \\
4 & + & x & 2 & x \; \\
5 & \text{jump}_to & 2 & & \\
\hline
\end{array}
\]

\[
S_3 = \{
\begin{array}{l}
\text{pop exp;} \\
\text{pop id;} \\
\text{generate_tuple(} \\
\quad "{+","id ",",} \\
\quad \text{ex ",","id ");} \\
\text{generate_tuple(} \\
\quad "{jump_to","i","","} \\
\text{c[j+1][2]:=next_quadruple();} \\
\}
\end{array}
\]

Quadruples: loops

Examples

\[
\text{do } x \{ S_0 \} := 1 \{ S_1 \}, \ 10 \{ S_2 \}, \ 2 \{ S_3 \} \\
\quad z := 1; \\
\quad y := 2; \\
\text{end}
\]

\[
\begin{array}{|c|c|c|} 
\hline
\text{c} & \text{i} & \text{j} & \text{exp} & \text{id} \\
\hline
1 & := & 1 & x \; & \\
2 & \text{jump}_> & x & 10 & \\
3 & \text{jump}_to & & & \\
4 & + & x & 2 & x \; \\
5 & \text{jump}_to & 2 & & \\
\hline
\end{array}
\]

\[
\text{Examples}
\]

\[
\begin{array}{|c|c|c|} 
\hline
\text{c} & \text{i} & \text{j} & \text{exp} & \text{id} \\
\hline
1 & := & 1 & x \; & \\
2 & \text{jump}_> & x & 10 & \\
3 & \text{jump}_to & & & \\
4 & + & x & 2 & x \; \\
5 & \text{jump}_to & 2 & & \\
\hline
\end{array}
\]

\[
S_3 = \{
\begin{array}{l}
\text{pop exp;} \\
\text{pop id;} \\
\text{generate_tuple(} \\
\quad "{+","id ",",} \\
\quad \text{ex ",","id ");} \\
\text{generate_tuple(} \\
\quad "{jump_to","i","","} \\
\text{c[j+1][2]:=next_quadruple();} \\
\}
\end{array}
\]
Quadruples: loops

Examples

\[
\begin{align*}
do & \ x \ (S_0):=1(S_1), \ 10(S_2), \ 2(S_3) \\
& \quad \ z:=1; \\
& \quad \ y:=2 \\
& \end{do}
\]

```
1  :=  1  x  \\
2  :=  x  10  \\
3  :=  6  \\
4  :=  x  2  x  \\
5  :=  2  x  \\
6  :=  1  \\
7  :=  2  y
```

S_3={
  generate_tuple( "(jump_to," ,j+2, ",,)" );
  c[j][2]:=next_quadruple();
}

Examples

\[
\begin{align*}
do & \ x \ (S_0):=1(S_1), \ 10(S_2), \ 2(S_3) \\
& \quad \ z:=1; \\
& \quad \ y:=2 \\
& \end{do}
\]

```
1  :=  1  x  \\
2  :=  x  10  \\
3  :=  6  \\
4  :=  x  2  x  \\
5  :=  2  x  \\
6  :=  1  \\
7  :=  2  y
```

S_3={
  generate_tuple( "(jump_to," ,j+2, ",,)" );
  c[j][2]:=next_quadruple();
}
Examples

```plaintext
do x {S0} := 1{S1}, 10{S2}, 2{S3}
z := 1;
y := 2
end {S5}
```

```
S5 =
generate_tuple("(jump_to, j+2, ",")");
c[j][2] := next_quadruple();
```

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.