9 Intermediate Code Generation

9.1 Introduction

**Place of this topic in relation to course:**

- Previously, we looked at ‘direct’ code generation, where assembler or machine code was generated directly from the parse tree.
- Now we will look at code generation which is performed in two steps:
  1. Generate an internal representation from the parse tree
  2. Map from this to the target code language
**Value of Intermediate Code Generation:**

- Typically the compiler needs to produce machine code or assembler for several target machines.
- The ‘intermediate code’ representation is neutral in relation to target machine, so the same intermediate code generator can be shared for all target languages.
- Less work in producing a compiler for a new machine.
- Also, it is easier to optimise the intermediate code representation, and the optimisation is machine independent.

**Main Methods of Intermediate Code generation:**

- Two main forms used for representing intermediate code:
  
  1. *Postfix Notation*: the abstract syntax tree is linearised as a sequence of data references and operations on these.
     - For instance, the tree for: `a * ( 9 + d )`
     - Can be mapped to the equivalent postfix notation:
       - `a9d+*`

  2. *Quadruples*: All operations are represented as a 4-part list:
     - `(op, arg1, arg2, result)`
     - E.g., `x := y + z` -> `(+ y z x)`
9 Intermediate Code Generation

9.2 Postfix Notation

Postfix Notation:

- Also called ‘suffix notation’ and ‘reverse polish’.
- The source language statement:
  \[ a \times (9 + d) \]
- … can be rewritten as:
  \[ a9d+* \]
- Any expression can be written unambiguously without parentheses, nor need for stating operator precedence.
- We can easily build interpreters for these expressions, using a stack.
- This is the procedure followed by most assemblers.
Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

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

Introductory examples: evaluation of expression in inverse polish notation

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

1. 2. 
3. 4. 
5. 6. 
7. 8. 
9. 10.

d
9
a
Intermediate Representations

Introductory examples: evaluation of expression in inverse polish notation

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

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

Introductory examples: evaluation of expression in inverse polish notation

\[ a \times ( 9 + d ) \Leftrightarrow a_9d+ \]
Intermediate Representations
Introductory examples: evaluation of expression in inverse polish notation

\[ a \times (9 + d) \iff a_9d+* \]

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

Intermediate Representations
Introductory examples: evaluation of expression in inverse polish notation

\[ a \times (9 + d) \iff a_9d+* \]

\[ a \times (9+d) \]
9 Intermediate Code Generation

9.3 Generating Postfix Form from the Parse tree

How to generate the postfix code:

- A ‘semantic stack’ is used to represent the postfix code being generated.
- This stack is initially empty.
- Semantic actions are connected with each production (as seen in semantic analysis).
- Only one semantic action is used to create the semantic stack:
  - push <value>: place a value (address or operator) on the semantic stack
Intermediate Code Generation
Generating Postfix Code

How to generate the postfix code:

• All logic or arithmetic operations are assumed to be directly supported by the machine:
  
  +, *, /, -, and, or

• Later we will look at how other structures are generated:

Intermediate Code Generation
Generating Postfix Code

How to generate the postfix code:
• An example ‘semantic grammar’

  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)
How to generate the postfix code:

- An example ‘semantic grammar’

```
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)
```

Parentheses have no effect on resulting postfix code
Se puede comprobar que las siguientes acciones semánticas:

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

Generación de la representación sufija asociada al programa

- push a
- push b
- push c
Se puede comprobar que las siguientes acciones semánticas:

\[ E \rightarrow E + T \]
\[ E \rightarrow E - T \]
\[ T \rightarrow T \times F \]
\[ T \rightarrow T / F \]
\[ T \rightarrow F \]
\[ F \rightarrow i \]
\[ F \rightarrow (E) \]
\[ F \rightarrow -F \]

Generación de la representación sufija asociada al programa:

```
push a
push b push c
push a
push /
```
Se puede comprobar que las siguientes acciones semánticas:

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow -F$

Generación de la representación sufija asociada al programa:

push $a$

push $b$

push $c$

push $a$

push $/$

push $+$

push $*$

push $-$

push $+$

push $/$

push $a$

push $b$

push $c$
Suffix notation

Generation of the suffix notation associated to a program

The Result

• We can treat the stack generated by the prior process as the intermediate representation of the input, in postfix notation form:

a b c a / + *

9 Intermediate Code Generation

9.4 Extending to other structures
Extending to other structures:

- It is clear postfix notation can be used for representing mathematical expressions.
- Can it also be extended to handle all other programming constructs?
  - E.g., assignment? Control-flow structures?
- If feasible, this notation would offer a good candidate for intermediate code representation:
  - It is simple
  - It is easy to interpret
  - It is unambiguous

Unary Operators:

- Some operators, such as ‘-‘, can be a unary (single argument) or binary operator.
- If we just map these operators into the postfix notation, it will be ambiguous whether they operate on one or two arguments.
- Two solutions:
  1. Convert to a binary op:
     - Map: ‘-a’ to ‘0a-’
  2. Create a new operator for the unary use of ‘-‘:
     - map: ‘-a’ to ‘a_’
How to generate the postfix code:

- An example ‘semantic grammar’
  
  $E \rightarrow E + T \quad \text{(push +)}$
  
  $E \rightarrow E - T \quad \text{(push -)}$
  
  $E \rightarrow T$
  
  $T \rightarrow T * F \quad \text{(push *)}$
  
  $T \rightarrow T / F \quad \text{(push /)}$
  
  $T \rightarrow F$
  
  $F \rightarrow i \quad \text{(push i)}$
  
  $F \rightarrow (E)$
  
  $F \rightarrow -F \quad \text{(push _)}$

Note that Unary – has distinct operator

Assignment:

- An assignment statement:
  
  $V = a + b$
  
  …can be represented as follows:
  
  $V_{ab+}$
  
  The ‘=’ operator thus:
  
  - Takes the previous two elements on the stack
  - Assumes the earlier one is a memory address
  - And places the second value in that location.

  $a = a * (9 + d) \iff aa9d+*=$
Flow Control:

- Many flow control statements (if-then, while, for, etc.) can be mapped onto assembler which depends on some sort of conditional or unconditional jump statement:
  \[ \text{CMP} \ [A], \ [B] \]
  \[ \text{JZ} \ L1 \]
- The intermediate code can make use of the same kind of solution:
  - label jmp_to
  - Jmp_to is a unary operator which takes the prior element on the stack as a memory address.
  - Similar for: \textit{jump_if_zero} and \textit{jump_if_false}

If Statement

Stmt ::=- if LExpr then #A1 Block #A2 else Block #A3
#A1: Generate code to jump if exp non-zero
#A2: At end of then code, generate jmp to end
Also generate label for start_else
#A3: Generate label for end_ifelse

#A1 := \lambda \{ Stmt.startElse= GenLab ;
push Stmt.startElse ;
push JNZ \}

#A2 := \lambda \{ Stmt.endlab= GenLab ;
push Stmt.endlab ;
push JMP ;
push Stmt.startElse ;
push GENLAB \}

#A3 := \lambda \{ push Stmt.endLab ;
push GENLAB \}
9 Intermediate Code Generation

9.5 Generating Target Code from Postfix form

Generating Assembler from Postfix form: stack-based assembler

- The operators used in postfix notation are similar to those used in most assembler languages.
- The stack data structure is also a basis for most assemblers
- It should thus be easy to translate from postfix representation to a given assembler language.
Generating Assembler from the postfix stack:

- For any **operand**:
  - Generate code to push it onto the stack:
    
    ```
    push dword [a]
    ```

- For any **operator**:
  - Pop from the stack as many operands as arguments the operator requires.
  - Perform the operation.
  - Push the result.
    ```
    pop eax
    pop ebx
    idiv ebx, eax
    push ebx
    ```

---

**Our example:**

```
push dword [a]
```
Our example:

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

Intermediate Code Generation

Code generation using intermediate representation

In our example:

```
push dword [a]
push dword [b]
push dword [c]
```
In our example:

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

pop  eax
pop  ebx
idiv ebx, eax, eax
push ebx
```

```
push dword [a]
push dword [b]
push dword [c]
push dword [a]
pop  eax
pop  ebx
idiv ebx, eax
push ebx
push ebx
```
In our example:

```
push dword [a]
push dword [b]
push dword [c]
push dword [a]
pop  eax
pop  ebx
idiv ebx, eax
push ebx
pop  eax
pop  ebx
add ebx, eax
push ebx
```
Intermediate Code Generation
Code generation using intermediate representation

Example: Assignment op:

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

The ‘a’ on the LHS of an assignment statement represents an address (where to store the final result)

Suffix notation
Code generation using intermediate representation

```
push dword a
```
Each ‘a’ on the RHS of an assignment statement represents a value (where to get the value from).

For this reason, the code generated from a LHS identifier is different from those on the RHS.
Suffix notation
Code generation using intermediate representation

Each ‘a’ on the RHS of an assignment statement represents a value (where to get the value from).

For this reason, the code generated from a LHS identifier is different from those on the RHS.

When we generate the intermediate representation, we need to distinguish the two cases.

Suffix notation
Code generation using intermediate representation

push dword a
push dword [a]

push dword a
push dword [a]
push dword [b]
push dword [a]
pop  eax
pop  ebx
idiv ebx
push eax
pop  eax
pop  ebx
add  eax, ebx
push eax
pop  eax
pop  ebx
mul  ebx
push eax
pop  ebx
pop  eax
mov  dword [eax], ebx
Suffix notation

Code generation using intermediate representation

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

Code generation using intermediate representation

\[ a_i \cdot a \cdot b \cdot c \cdot a \div a + a^* = \]

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

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

push eax
pop eax
pop ebx
add ebx, eax
push ebx
Suffix notation
Code generation using intermediate representation

\[
\begin{array}{cccccc}
  \text{a}_1 & a & b & c & a & / + * = \\
\end{array}
\]

\[
\begin{array}{cccccc}
  \text{push dword a} \\
  \text{push dword [a]} \\
  \text{push dword [b]} \\
  \text{push dword [c]} \\
  \text{push dword [a]} \\
  \text{pop eax} \\
  \text{pop ebx} \\
  \text{idiv ebx, eax} \\
  \text{push ebx} \\
  \text{pop eax} \\
  \text{pop ebx} \\
  \text{add ebx, eax} \\
  \text{push ebx} \\
  \text{pop eax} \\
  \text{pop ebx} \\
  \text{mul ebx, eax} \\
  \text{push ebx} \\
\end{array}
\]
Exercise

1. Convert the following program into Postfix notation:

   $$ \text{a} = (\text{-a} + 2\text{b})/\text{a} $$

2. Produce the assembly code corresponding to this example.