Compiling Techniques CSM201

Kristian Guillaumier

http://www.cs.um.edu.mt/~kguil

kguil@cs.um.edu.mt

Programmable Machines

- Processors are programmable in a language called **Machine Code**.
- The range of features available to this language is defined by the **Instruction Set** of the processor.
- Although the instruction set actually contains primitive, basic operations you can actually write **any program** using it.
- Each processor family has it's own instruction set. Though basic operations are common to most of them, they're incompatible with each other in almost every other respect.

A Simple Program in Machine Code (1)

- Consider the following simple statement in BASIC:

  length = 2*(side1-side2) + 4*(side3-side4)

- We'll write the equivalent machine code to execute this statement for a processor with a limited instruction set called SIMPLE.
- Variables and their values are stored in memory (RAM).
- In addition the SIMPLE processor has a single memory location (register) called the accumulator.

A Simple Program in Machine Code (2)

Memory Map

Memory Location	Variable/Constant	Value
Loc1	Length	Unknown at startup
Loc2	Side1	6
Loc3	Side2	3
Loc4	Side3	4
Loc5	Side4	2
Loc6	2	2
Loc7	4	4
Loc8	Temp	Unknown at startup

A Simple Program in Machine Code (3)

Instruction Set

Instruction	**Meaning**
Load <addr>	Loads a value from the memory location <addr> into the accumulator.
Sub <addr>	Subtracts a value read from <addr> from the one in the accumulator. The result is stored in the accumulator.
Add <addr>	Like above but adds.
Mul <addr>	Like above but multiplies.
Store <addr>	Stores the result from the accumulator into the memory location <addr>.

A Simple Program in Machine Code (4)

- The machine code to evaluate the expression would be:

```
Load    Loc2
Sub     Loc3
Mul     Loc6
Store   Loc8    ← we need a temp variable
Load    Loc4
Sub     Loc5
Mul     Loc7
Add     Loc8
Store   Loc1
```

Advantages and Disadvantages

- Advantages:
  - Programmers are required to have an intimate knowledge of the processor. This can lead to highly optimised code.
  - May be the only way to program the processor (like an embedded processor in a microwave oven).
- Disadvantages:
  - Programmers are required to have an intimate knowledge of the processor. Difficult to learn.
  - Development time takes longer.
  - ‘Easier’ to make mistakes.
  - Not portable. The program is tied down to the processor it was written for.
  - Human beings ‘think’ about algorithims differently than a processor does.

What is a Compiler?

- Informally a Compiler:
  - Translates a program in a language (*source language*) to another language (*target language*) usually machine code.
  - Checks for syntactical correctness.
  - Checks for semantic correctness.

Cousins of the Complier (1)

- Assemblers
  - Similar to compilers (translation/checks syntax/etc…) but the source language is Assembly Language.
- Cross-Compilers
  - The compiler program runs on a processor type, but the machine code it produces is designed to run on a different one. An example of using a cross compilers is to develop software that runs on mobile phones.
  - Cross Compilers are useful:
    - Either because the target machine doesn't have a compiler of it's own.
    - Or because it doesn't have the resources to run the compiler in the first place.

Cousins of the Complier (2)

- Interpreters
  - An interpreter translates a program into a lower level version of it, but it still cannot run directly on the processor. It depends on some runtime support. Examples of interpreted language include:
    - Command Line Interpreters (BASH, command.com)
    - Batch Files
    - VBScript, JavaScript
  - Execution is slower since the translation occurs each time the program is executed.
  - Interpreters are easier to write.

A Deeper Look into Compilers

- To keep things manageable the process of compilation is separated into 3 distinct (though connected) phases:
  - Lexical Analysis
  - Syntax and Semantic Analysis
  - Code Generation

Lexical Analysis

- Lexical Analysers are also called Scanners.
- Recall that a program is made up of many small entities:
  - Keywords: IF, THEN, ELSE, …
  - Identifiers: counter, my_var, openfile
  - Numbers
  - Symbols: +, /, >, >=
- Put Simply, the scanners job is to:
  - Open the source file,
  - Recognise the entities and represent them as tokens,
  - Remove Comments,
  - Produce error reports.

Example

```
for counter = 1 to 20 print "hello world" next
```

Tokens

for	Keyword	
counter	Identifier	Name = Counter
=	Operator	Equals
1	Constant	Value = 1
to	Keyword	
20	Constant	Value = 30
print	Keyword	
“hello world”	String	Value = “hello world”
next	keyword	

Syntax Analysis

- The syntax analyser is also known as the **Parser**.
- For all the compiler is concerned, the sequence of tokens produced by the scanner is just a random sequence of symbols. It is the job of the syntax analyser to ensure that these symbols are structured correctly according to the definition of the language. For example:
  - Every BEGIN must match an END in Pascal.
  - Every statement must end in a semi-colon.

Semantic Analysis

- Even though the structure of the language is correct, the MEANING of the statements may be invalid according to the semantics of the language.

  my_var = my_var + 1

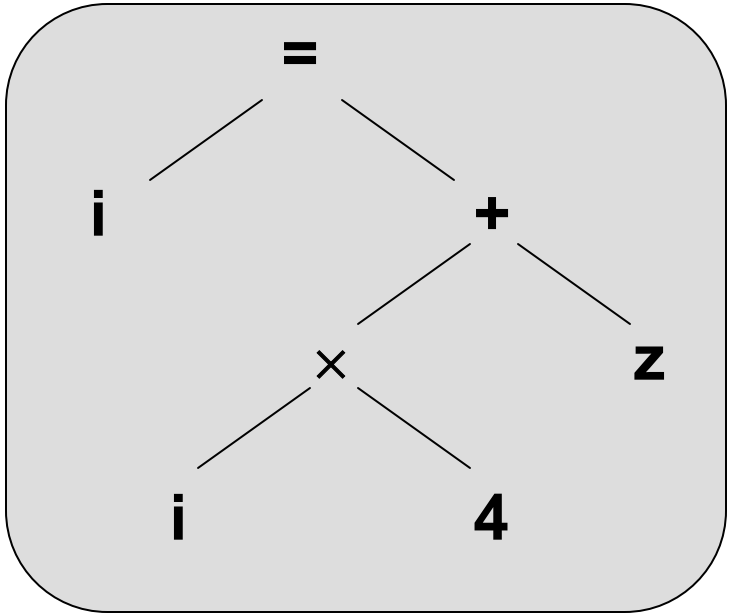
  is correct in terms of syntax, but is my_var is declared as a string, the arithmetic addition of a number to a string isn't really correct.

The Symbol Table and Parse Tree

- The output produced by the Syntax and Semantic Analyser is the:
  - Symbol Table:
    - Stores information about identifiers and functions, such as their types, sizes, names, number of arguments etc…
  - Parse Tree:
    - Stores the structure of the program.

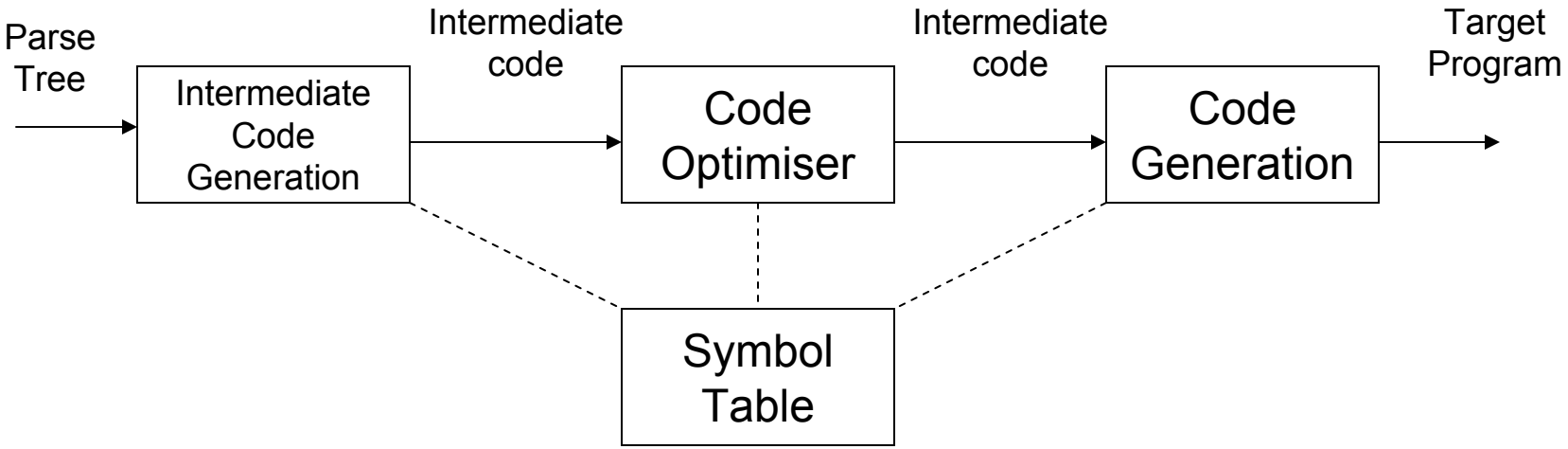
Example Parse Tree

- i = (I * 4) + z



Code Generation

- After all the preceding phases have been completed successfully without errors, the compiler will proceed to build the target code from the data structures previously constructed.
- In many cases Code Generation of further split:



Intermediate Code Generation

- The code generator starts off by generating an intermediate form of code representation before actually building the target code.
- The main difference between intermediate code and the actual target code is that certain details such as the exact memory locations are omitted.
- A common representation format for intermediate code is the **Three-Address Code**.

Three-Address Code

```
temp1 = 60
temp2 = id2 + temp1
id1 = temp2
```

Operator	ARG1	ARG2	Result
=	60		temp1
+	id2	temp1	temp2
=	temp2		id1

Code Optimisation

- This phase attempts to rearrange the code to obtain a smaller or faster running version.

```
temp1 = 60
temp2 = id2 + temp1
id1 = temp2
```

Equivalent to:

```
id1 = id2 + 60
```

Front and Back Ends (1)

- Commonly compiler phases are split into two different categories:
  - The front end: this stage is concerned with the phases related to the source language and are independent of the target. This part usually consists of the lexical analyser, syntax analysis, symbol table creation, semantic analysis and intermediate code generation.
  - The back end: consists of the stages dependent on the target machine. This usually consists of the code generation and certain parts of the optimiser.

Front and Back Ends (2)

- Splitting the compiler into front and backends has the following advantages:
  - The backend can be modularly changed to compile the same source language for a different platform.
  - Compilers for different source languages usually produce standard intermediate code and may reuse the same backend.

Language Specification

- Programming languages must be specified and properly described before attempting to write a compiler for them.
- The specification is written in a **meta-language**.
- Meta-Languages need to be unambiguous and we rely on **Formal Languages** to assist.

Formal Languages Primer (1)

- In order to specify a formal language rigorously we need to introduce some concepts:
- A **Symbol** or **Token** is an atomic (indivisible) entity usually a character, digit or keyword.
- An **Alphabet**, denoted by Σ, is the finite, non-empty set of symbols.
- A **String** over the alphabet is a sequence $a_1a_2\ldots a_n$ of symbols from Σ.
- The symbol ε denotes the empty string.
- $\varepsilon a = a\varepsilon = a$

Formal Languages Primer (2)

- The set of all strings over the alphabet Σ, including the empty string ε, is denoted by the **Kleene Closure** - Σ^*.
- The set of all strings over Σ, whose length is at least 1 (i.e. does not contain ε), is denoted by the **Positive Closure** - Σ^+.
- A **Language** L over the alphabet Σ is a subset of Σ^*.

Regular Expressions

- Many languages (though not all) may be described using a notation called **Regular Expressions**.
- Regular expressions specify strings in a language by using symbols from it's alphabet and a few special meta-symbols:
  - **Concatenation:** when we wish to concatenate symbols or string we write them next to each other of use the . (dot) meta-symbol for extra clarity.
  - **Alternation:** when there is a choice between to symbols α and β, they are separated by the | (bar) symbol.
  - **Repetition:** a symbol α followed by a * (star) indicates that there are zero or more repetitions of α.
  - **Grouping:** a group of symbols may be grouped by surrounding them by the meta-symbols (and) – parenthesis.

An Example

- Consider the expression:

```
1 ( 1 | 0 )* 0
```

- This expression represents all the strings that start with a 1, end in a 0 and have an unlimited (possibly empty) number of 1's and 0's in between.

```
{10, 100, 110, 1000, …}
```

Notes on Regular Expressions

- Precedence from highest to lowest: Parenthesis → Repetition → Concatenation → Alternation

  ab* ≠ (ab)*

- If the meta-symbols are part of the alphabet, they should be enclosed in quotes. For example, comments in Pascal would be:

  "(" "*" c* "*" ")"

  where $c \in \Sigma$

- Another convention normally used is that of the + repetition instead of the *. It has the same meaning as the Positive Closure. Basically it's a shortcut for writing aa*

Algebraic Properties of Regular Expressions

`A	B = B	A`	Commutativity for alternation
`A	(B | C) = (A | B)	C`	Associativity for alternation
`A	A = A`	Absorption of alternation
`A . (B.C) = (A.B).C`	Associativity for concatenation
`A (B	C) = AB	AC`	Left distributivity
`(A	B) C = AC	BC`	Right distributivity
`A ε = ε A = A`	Identity for concatenation
`A* A* = A*`	Absorption for closure

Regular Expressions

- Description of Identifiers

  `(_|A| … |Z|a| … |z).(_|A| … |Z|a| … |z|0| … |9)*`

- Description of Integers

  `(0 | 1 | 2 | … | 9)+`

Grammars

- Formally a **Grammar** is a quadruple {N,T,P,S} where:
  - N is the finite set of non-terminal symbols,
  - T is the finite set of terminal symbols (Σ),
  - P is the finite set of production (or grammar) rules,
  - S is the starting, goal or sentence symbol.
- A sentence is a string entirely composed of terminal symbols.

Example

- Consider the following language:

ε
ab
aabb
aaabbb
aaaabbbb
...

- The grammar for the above language is:

({S}, {a,b}, P, S)
where P is:
S → ε
S → aSb

Another Example

- Consider the following rules for a context sensitive grammar:

```
1) S  → aSBC
2) S  → aBC
3) CB → BC
4) aB → ab
5) bB → bb
6) bC → bc
7) cC → cc
```

- The following is a *derivation* from S to a valid string:

```
S
aSBC    (by rule 1)
aaBCBC  (by rule 2)
aaBBCC  (by rule 3)
aabBCC  (by rule 4)
aabbCC  (by rule 5)
aabbcC  (by rule 6)
aabbcc  (by rule 7)
```

Types of Grammars

- The complexity and structures of the rules in a grammar determines what types of languages we can describe and recognise using it. These different “Grammar Types” are categorised by the **Chomsky Hierarchy**:
  - Type 0 – Unrestricted Grammars
  - Type 1 – Context Sensitive Grammars
  - Type 2 – Context Free Grammars
  - Type 3 – Regular Grammars

Unrestricted Grammars

- Productions take the form:

$$A \rightarrow \alpha$$

where,

  - A and α are arbitrary symbols in the vocabulary $\mathbf{N} \cup \mathbf{T}$.

Context Sensitive Grammars

- Productions take the form:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

where,

- $A \in N$
- $\gamma \neq \varepsilon$
- $\alpha, \beta, \gamma \in (N \cup T)^*$
- May also include the rule $S \rightarrow \varepsilon$

Context Free Grammars

- Productions take the form:

$$A \rightarrow \alpha$$

where,

- A is a single non-terminal symbol ($A \in N$),
- α is a, possibly empty, string of terminals and/or non-terminals.

Regular Grammars

- Productions take the form:

(i)	-or-	(ii)
$A \rightarrow \alpha$		$A \rightarrow \alpha$
$A \rightarrow \alpha B$		$A \rightarrow B\alpha$

The difference between the two is that one is right recursive (since B can be equal to A) and the other is left recursive. Regular grammars must either be one or the other, but never both (otherwise this would be a type 2 grammar)

Backus-Naur-Form (BNF)

Symbol	Meaning
::=	**'is defined as'**
|	**Or**
< >	**Angles brackets surround category symbols – These symbols are called *Non-Terminals***
[ ]	**Optional items are surrounded by square brackets**
{ }	**Zero or more repetitions of an item are surrounded by curly brackets**
" "	**Exact symbols in the language are enclosed in quotes – These symbols are called *Terminals*. Sometimes the quotes are omitted.**
;	**End of line**
ε	**Empty or Nothing**

BNF By Example (1)

Consider the identifier:

my_variable

In plain English:

Identifiers consist of any sequence alpha-numeric characters and the underscore symbols. However an identifier cannot start with a digit.

Formally in BNF:

```
<ident> ::= <alpha> | "_" {<alpha>|<digit>|"_"};

<alpha> ::= "a"|"b"|…|"z"|"A"|"B"|…|"Z";

<digit> ::= "0"|"1"|…|"9";
```

BNF by Example (2)

Problem:

Construct a BNF specification for simple expressions limited to integer numbers and identifiers. The operators allowed in this type of statement are + and –.

For Example:

```
3

counter

counter + 1

counter + (1 – y)

3 + 4
```

BNF by Example (2½)

```
<expr> ::= <factor> | <factor> <op> <expr>;

<factor> ::= <integer> | <ident> | "(" <expr> ")";

<op> ::= "+" | "-";

<integer> ::= <digit> {<digit>};
```

Extensions to BNF (EBNF)

- In order to improve the readability and conciseness of descriptions in BNF several extensions have been proposed to the notation. BNF with these extensions is called EBNF.
- *Kleene Cross*: a sequence of *one or more* items of a class are:

```
<unsigned-int> ::= <digit>+
```

- *Kleene Star*: a sequence of *zero or more* items of a class are:

```
<ident> ::= <letter><alphanumneric>*
```

- Braces are used for *grouping* instead of the usual ‘zero or more’ interpretation.

```
<ident> ::=  <letter>{<letter>|<digit>}*
```

An Example in EBNF

- Consider BNF for variable declarations in Basic:

```
<var-decl> ::= "dim" <var-decl-list>

<var-decl-list> ::=
     <var-decl-item> {ε | "," <var-decl-list>}

<var-decl-item> ::= <ident> "as" <var-type>
```

- In EBNF could would be written as:

```
<var-decl> ::= "dim" <var-decl-list>

<var-decl-list> ::=
     <var-decl-item> {"," <var-decl-item>}*

<var-decl-item> ::= <ident> "as" <var-type>
```

Designing a Lexical Analyser

- The key function in a lexical analyser is a routine called **GetNextToken** that extracts tokens one-by-one from the source file.
- The lexical analyser repeatedly makes calls to GetNextToken to process the whole file.
- When tokenising the input it is important to identify the **Token Separators**. These separators are special characters that delimit one token from an other. In many programming languages, the token separators are usially spaces, tab stops and carriage returns.
- The scanning loop can look like:

```
Initialise;
loop
     symbol = GetNextToken();
     Print symbol;
Until symbol = End_Of_File;
Clean_Up;
```

Recognising Tokens

- The scanner will start recognising a token after reading the first character:
- If the first character is:
  - A letter: then we're dealing with a keyword or identifier.
  - Numeric: then we're dealing with a number.
  - An Opening Quote (“): then we're dealing with a string.
  - Etc…

GetNextToken

```
Function GetNextToken
  CurrentChar = Get the next significant character

  If CurrentChar = EOF Then
        Deal with the end of file
  Else
        If CurrentChar = Digit Then
                CurrentToken = Deal With Number
        Else If CurrentChar = Letter Then
                CurrentToken = Deal With a Word
        Else If
                ...
        Else
                CurrentToken = Error - Illegal Character
        End IF
        GetNextToken = CurrentToken
   End If
End Function
```

GetWord

```
Function GetWord

  MyToken = CurrentChar
  CurrentChar = Get the next significant character

  While CurrentChar is Valid in a Word
      MyToken = MyToken + CurrentChar
      CurrentChar = Get next significant character
  Wend

  Return MyToken


End Function
```

Look Ahead

- Since we are reading characters until we find one that is not part of a word, the last one is essentially one extra character. We must not discard it. In fact *CurrentChar* is usually implemented as a global variable.
- This extra character is called the *LookAhead* character.

GetString

```
Function GetString

  MyToken = CurrentChar ' the opening quote
  CurrentChar = Get the next significant character

  While CurrentChar is not the Closing Quote
        MyToken = MyToken + CurrentChar
        CurrentChar = Get next significant character
  Wend

  MyToken = MyToken + CurrentChar ' The closing quote

  ' Again read an extra character for consistency
  CurrentChar = Get the next significant character

  Return MyToken

End Function
```

Error Reporting

- There are only a few errors that can be detected by the scanner. Such errors include:
  - Missing closing quote in a string. Missing quotes are a major issue, since characters will be read until the opening quote of the next sting are found, potentially ‘eating-up’ much of the actual code. This problem is typically alleviated by not allowing strings to span over multiple lines.
  - Illegal characters in the input file.

Parsing

- When analysing a programs syntax, a data structure called the **Parse Tree** is built to reflect the structure of the program.
- The nodes of the parse tree are the Non-Terminal symbols, whilst the leaves are the Terminals (Σ).
- The root node is the sentence symbol (S).
- There are two main methods of parsing:
  - Top-Down Parsing – the parse tree is build from the root downwards.
  - Bottom-Up Parsing – the parse tree is built from the leaves upwards to the root.

Simple Parsing Example

- Consider the BNF specification for a simple assignment statement:

```
<assign>      ::= <lhs> <ass-op> <rhs>;
<lhs>         ::= <ident>;
<ass-op>      ::= "=";
<rhs>         ::= <integer>
                    <arith-op> <integer>;
<arith-op>    ::= "+" | "-" | "*" | "/";
```

Top-Down Parse (1)

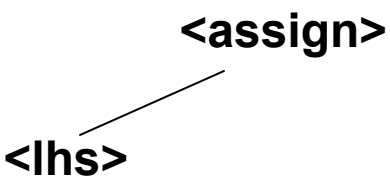
- In this example we will construct a parse tree for the assignment:

```
counter = 3 + 4
```

- The parser looks for the sentence symbol to create the root node:

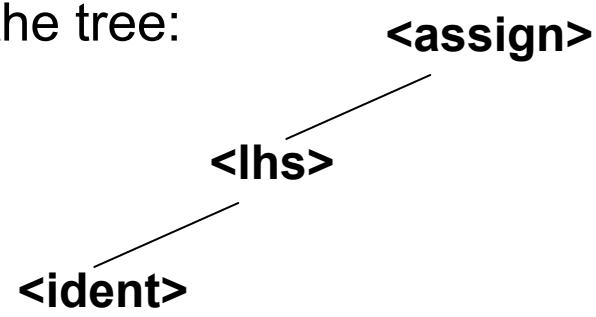
<assign>

- The first symbol in the rule is an LHS so we add it to the parse tree:

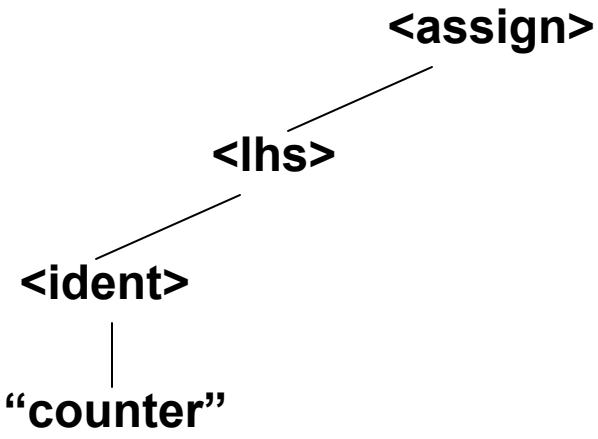


Top-Down Parse (2)

- An LHS is an IDENT which we add to the tree:

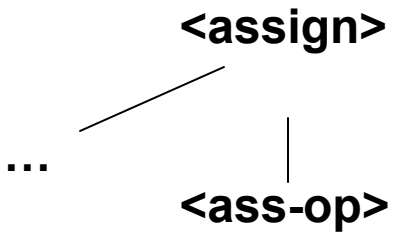


- In our example “counter” is an IDENT – we have a match and add it to the tree:

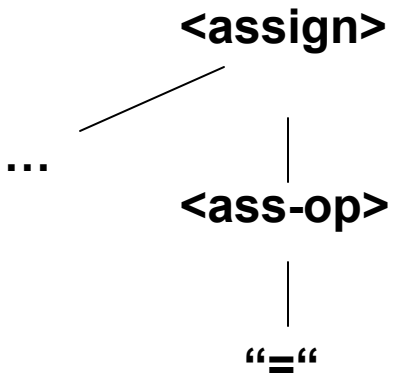


Top-Down Parse (3)

- The next expected item is the assignment operation, so it is added to the tree (at the current root because it is a non-terminal):

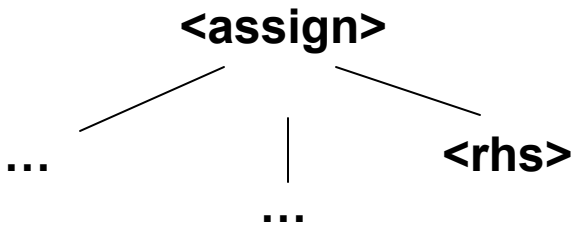


- An assignment operator is a non-terminal an looks at the next token and finds one:



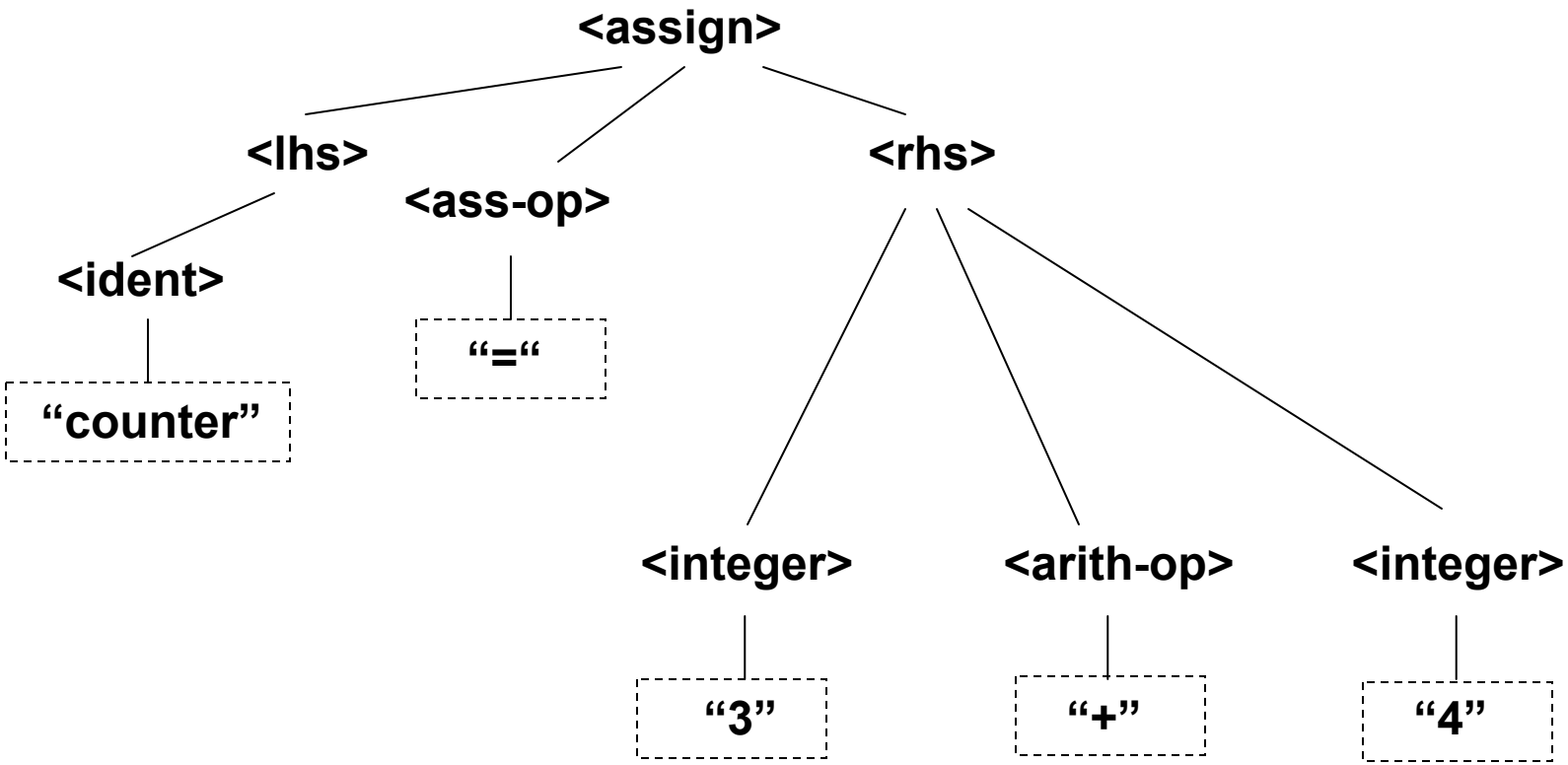
Top-Down Parse (4)

- The next item to be expected is an RHS which is a non-terminal so we add it to the current root:



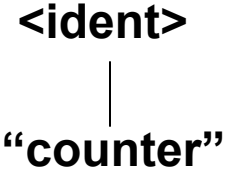
- The procedure is repeated until we complete the tree and find out that our assignment is structurally correct.

Top-Down Parse (5)

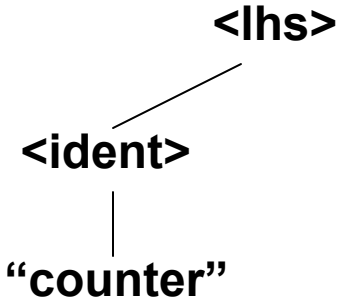


Bottom-Up Parse (1)

- The first token in the input is an IDENT so the leaf of the tree is obtained:

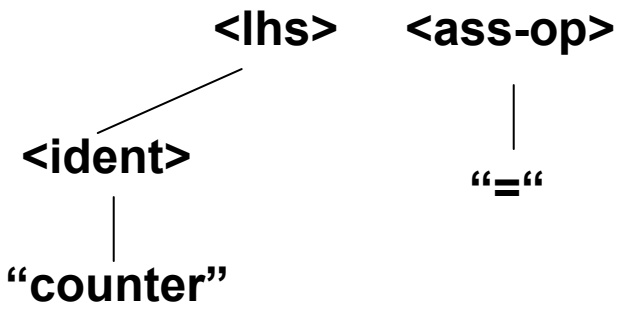


- By having a look at the rules we see that an IDENT is an LHS, so the tree grows up:



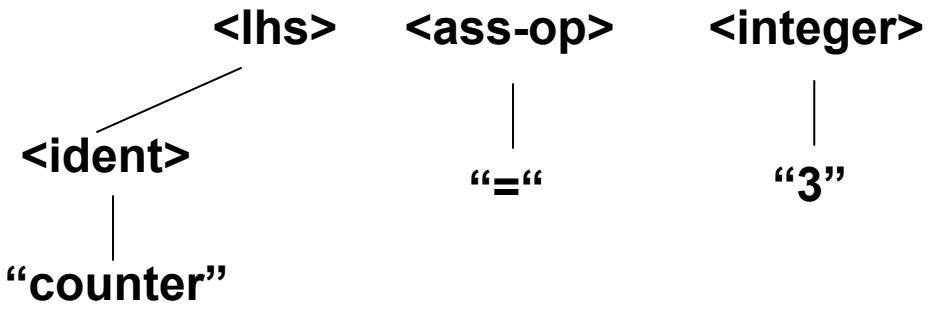
Bottom-Up Parse (2)

- An LHS on it's own cannot be resolved into anything else, so we continue reading from the input. We find an "=" sign so it's added as a leaf:

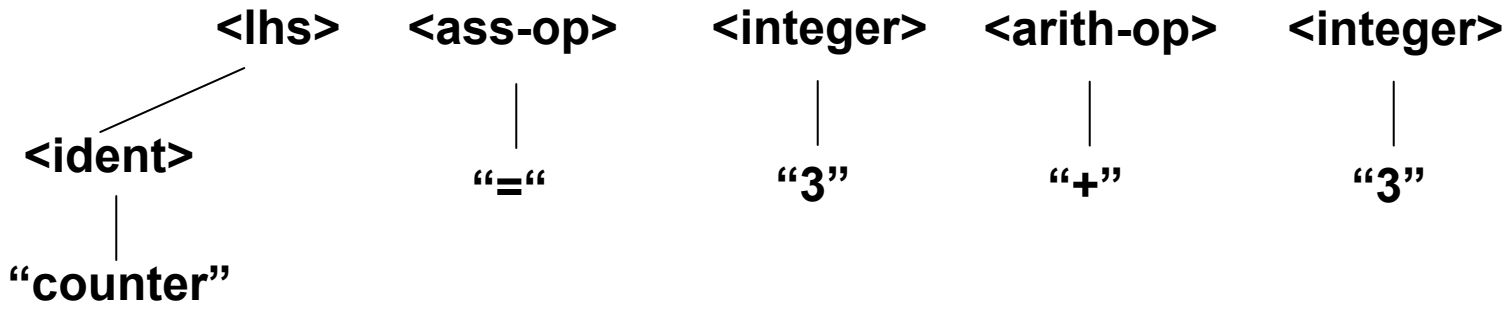


- The ASS-OP non-terminal cannot be resolved into anything else and neither can the LHS ASS-OP sequence so we continue reading the input and find a number which we add as a leaf:

Bottom Up Parse (3)

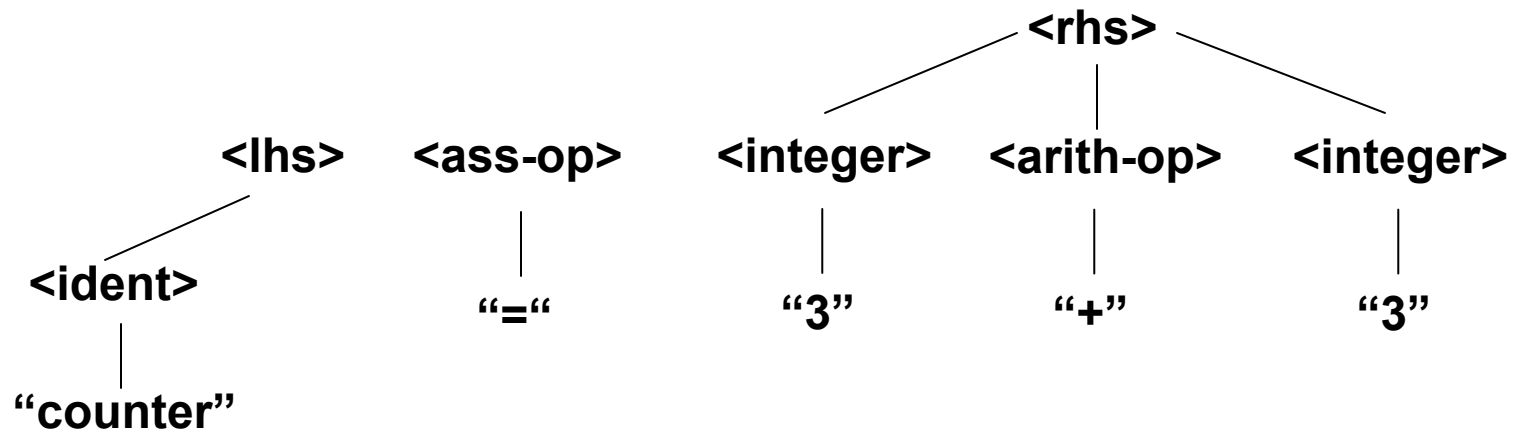


- This process continues until we consumed the whole input:



Bottom-Up Parse (4)

- After reading the last integer we see that the INTEGER ARITH-OP INTEGER sequence can be reduced to an RHS:



- Similarly in the next step we see that the resulting LHS ASS-OP RHS sequence can be further reduced to an ASSIGN, thus the parse is complete.

Things to Note…

- The grammar chosen for this example was purposely designed to keep the example simple.
- In reality parsing mechanisms are more sophisticated and grammars may really manifest properties that make parsing more complex. Consider the following grammar for a more elaborate assignment:

```
<assign>  ::= <lhs> <ass-op> <rhs>;
<lhs>     ::= <ident>;
<rhs>     ::= <factor> { <arith-op> <factor> };
<factor>  ::= <ident> | <integer>;
<ass-op>  ::= "="
<arith-op> ::= "+" | "-" | "*" | "/"
```

…Things to Note

- When parsing the assignment statement using the original grammar, when we read the identifier leaf, we saw that it could be reduced to an LHS (see Bottom-Up Parse (1)). Using the grammar presented above, we see that the identifier could be reduced to both an LHS or a FACTOR. The question here is – *Which path shall I follow?*
- Such issues are tackled by more sophisticated parsers.

Parsing a Variable Declaration (1)

- The simplest way to hand-code parsers is to provide programming language equivalents to BNF notational constructs:

Terminal Symbols	Test for the terminal symbol.
Non-Terminal Symbols	A procedure or function call.
Repetitions - { }	A while loop.
Alternatives - |	If-Then-Else statements.
Optional Items – [ ]	If-Then Statement.

Parsing a Variable Declaration (2)

- Consider simple variable declaration statements:

```
integer i,j,k
boolean isReady
```

- A suitable grammar to parse such statements would be:

```
<VarDecl>         ::= <TypeName> <VarNameList>;
<TypeName>        ::= INTEGER|BOOLEAN|REAL;
<VarNameList>     :: <VarName> {“,” <VarName>};
```

Parsing a Variable Declaration (3)

- For each Non-Terminal symbol, we define a function to parse it:

```
Function Parse_VarDecl(TOKEN)
  // Parse the type name
  LOOKAHEAD = Parse_TypeName(TOKEN)

  // Parse the variable name list
  LOOKAHEAD = Parse_VarNameList(LOOKAHEAD)

  Return LOOKAHEAD
End Function
```

Parsing a Variable Declaration (4)

- Since the Variable Declaration (VarDecl) is defined in terms of 2 other Non-Terminals (TypeName and VarNameList), it's parse is defined as calls to two other functions to parse each other non-terminal.
- Just as we had a look ahead character in the lexical analyser, we need a look ahead token in the parse tree.

Parsing a Variable Declaration (5)

- Parsing the Type Name:

```
Function Parse_TypeName(TOKEN)
  If TOKEN is INTEGER_TOKEN then
   Return NextToken // New Lookahead
  ElseIf TOKEN is BOOLEAN_TOKEN then
   Return NextToken
  ElseIf TOKEN is REAL_TOKEN then
   Return NextToken
  Else
   Print "Missing Type Name in Declaration"
   Return Error
  End If
End Function
```

Parsing a Variable Declaration (6)

- Note that in the previous example we used two of the transliteration mechanisms. We used simple checks to see if a token is a non-terminal and we used If-Then-Else statements for alternatives.
- Parsing the Variable Name List:

```
Function Parse_VarNameList(TOKEN)
  LOOKAHEAD = Parse_VarName(TOKEN)
  While LOOKAHEAD = COMMA_TOKEN
   LOOKAHEAD = NextToken
   LOOKAHEAD = Parse_VarName(LOOKAHEAD)
  End While

  Return LOOKAHEAD
End Function
```

Parsing a Variable Declaration (7)

- In the preceding example we used a combination of parsing both terminals (comma's) and non-terminals (variable names). The repetition was handled by a while loop that allowed for zero-or more items enclosed in the braces { }.
- Parsing the variable name:

```
Function Parse_VarName(TOKEN)
  If TOKEN = IDENT_TOKEN then
   Return NextToken
  else
   Print "Missing Identifier"
   Return Error
  End If
End Function
```

Parsing an If-Then Statement (1)

- Consider the following definition for an if-then statement:

```
<IF_STMT> ::= "IF" <EXPRESSION> "THEN"
                  <STMT_BLOCK>
                  ["ELSE" <STMT_BLOCK>]
                  "ENDIF"
```

```
Function Parse_If(TOKEN)
    If TOKEN = IF_TOKEN Then
          LOOKAHEAD = NEXTTOKEN
          // Parse the Expression
          LOOKAHEAD = Parse_Expression(LOOKAHEAD)
          If LOOKAHEAD = THEN_TOKEN then
                    LOOKAHEAD = NEXTTOKEN
                    LOOKAHEAD = Parse_StmtBlock(LOOKAHEAD)
                    // see if we have an else part
                    If LOOKAHEAD = ELSE_TOKEN then
                              LOOKAHEAD = NEXTTOKEN
                              LOOKAHEAD = Parse_StmtBlock(LOOKAHEAD)
                    End If
                    // get the ENDIF
                    If LookAhead = ENDIF_TOKEN Then
                              Return NEXTTOKEN
                    Else
                              Print "Missing ENDIF in conditional"
                              Return Error
                    End If
          Else
                    Print "Missing THEN in conditional"
                    Return Error
          End If

    End If
End Function
```

Syntax Errors

- Consider the following assignment statement with a missing comma:

```
integer i,j k;
```

- The parse will proceed normally in the VarNameList part until the variable k is found instead of the comma. The function will “think” that the variable name list has terminated returning k as the look ahead token. The parser will look for the ending semi-colon and will find a k instead reporting a missing semi-colon instead of the missing comma.

Backtracking (1)

- Consider the Grammar:

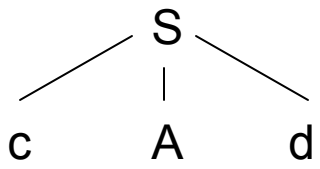
```
S → cAd
A → ab | a
```

- Given the input string **cad**, we try and construct the parse tree. We initially start by creating the Root of the tree from S (the current token is **c** in the input string:

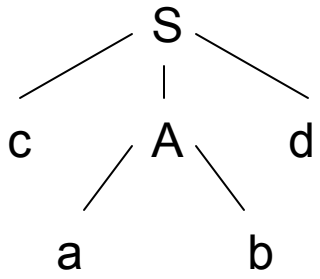
S

Backtracking (2)

- Clearly the current token **c** does not match **S** so we expand **S** using the first (and only rule):

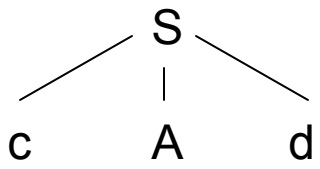


- The leftmost leaf matches our input symbol **c** so we proceed to the next one **a** and consider the next leaf **A**. In expanding **A**, we have two alternatives. Having no preference, we arbitrarily choose the first one to get the tree:



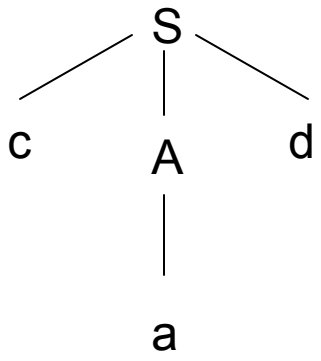
Backtracking (3)

- We have a match for the current token **a**, so we proceed with the next one **d**. Looking at the next leaf **b**, we see that the token does not match, so we must have expanded using the wrong production. We must backtrack to the state before the production was chosen – the current symbol is set back to **a**, and the tree:



Backtracking (4)

- We now try the other alternative and expand the tree to:



- The current input **a** matches the leaf. We move to the next token **d** and the next leaf, which match too. All the input has been consumed and we have completed the parse successfully.

Notes (1)

- The parsing methods we have seen are called **recursive-descent** parsers.
- Grammars can be rearranged to eliminate the need for backtracking. Parsers for such grammars are called **predictive parsers**.
- A left-recursive grammar (productions of the type A → Aα) can cause a recursive-descent parser to go into infinite loops, even if it has backtracking.
- Consider the grammar:

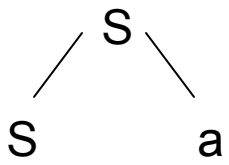
  S → Sa

  s → ε

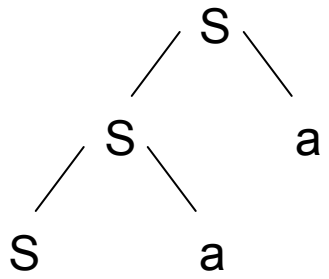
- Our task is to parse the string **aaaaaa**.

Notes (2)

- The current token is **a**. We expand the **S** node to get the tree:

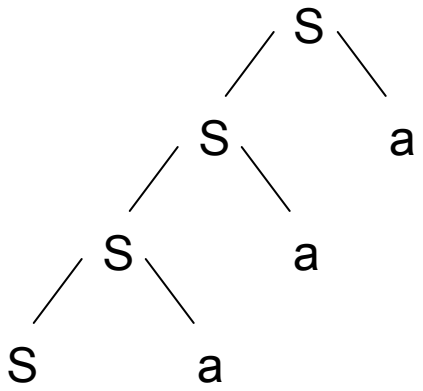


- The first leaf is a non-terminal so we expand again. We have two choices so we arbitrarily choose the first:



Notes (3)

- Again, the first leaf is a non-terminal so we expand again. We have two choices so we arbitrarily choose the first:



- The problem is that the tree will continue growing indefinitely without ever consuming any input (the terminating condition is never achieved).

Eliminating Left-Recursion

- A grammar is left-recursive if it has a derivation of the type $A \Rightarrow^{+} A\alpha$.
- As we have seen, top-down parsers cannot handle left-recursion, so we need a transformation these grammars into right recursive ones.
- A left-recursive production of the form

$$\mathbf{A} \rightarrow \mathbf{A}\alpha \mid \beta$$

- Can be rewritten as:

$$\mathbf{A} \rightarrow \beta\mathbf{A'}$$
$$\mathbf{A'} \rightarrow \alpha\mathbf{A'} \mid \varepsilon$$

Example

- Consider the following grammar for arithmetic expressions

```
E → E + T | T
T → T * F | F
F → ( E ) | Ident
```

- The grammar is rewritten as:

```
E  → TE'
E' → +TE' | ε
T  → FT'
T' → *FT' | ε
F  → ( E ) | Ident
```

Non-Immediate Left-Recursion

- Immediate left-recursion involves productions that involve left-recursive derivations in one step:

$$\mathbf{A \rightarrow A\alpha}$$

- There are cases where left-recursion may occur after more than one derivational steps. For example, the following grammar is not immediately left recursive:

$$\mathbf{S \rightarrow Aa \mid b}$$
$$\mathbf{A \rightarrow Ac \mid Sd \mid \varepsilon}$$

- The Non-Terminal S is left-recursive because $\mathbf{S \Rightarrow Aa \Rightarrow Sda}$

Multiple A-Productions

- In the previous example, we considered eliminating a single instance of left-recursions from an A-Production ($\mathbf{A} \rightarrow \mathbf{A}\alpha \mid \beta$)
- No matter how many left-recursive A-productions there are ($\mathbf{A} \rightarrow \mathbf{A}\alpha_1 \mid \mathbf{A}\alpha_2 \mid \ldots \mid \mathbf{A}\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$), all we have to do is replace the productions by:

$$\mathbf{A} \rightarrow \beta_1\mathbf{A}' \mid \beta_2\mathbf{A}' \mid \ldots \mid \beta_n\mathbf{A}'$$
$$\mathbf{A}' \rightarrow \alpha_1\mathbf{A}' \mid \alpha_2\mathbf{A}' \mid \ldots \mid \alpha_m\mathbf{A}' \mid \varepsilon$$

Algorithm for Eliminating Left-Recursion (1)

- This algorithm is guaranteed to eliminate left-recursion for grammars having:
  - No Cycles: $A \Rightarrow^{+} A$
  - No Empty Productions: $A \rightarrow \varepsilon$
- The input of the Algorithm is a grammar G with No Cycles or Empty Productions.
- The output of the Algorithm is the equivalent grammar without left-recursion.

Algorithm for Eliminating Left-Recursion (2)

Arrange the Non-Terminals in some order $A_1, A_2, \ldots, A_n$.

For i = 1 to n
  For j = 1 to (i-1)
    replace every production of the form
      $A_i \rightarrow A_j\gamma$ **by the productions**
      $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$, **where**
      $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
  Next j
  Eliminate the immediate left-recursion
    among the A_i productions
Next i

Example on a Grammar (1)

- Consider the grammar:

  S → Aa | b
  A → Ac | Sd | ε

- Note: technically, the algorithm is not guaranteed to work because of the A → ε empty production, though in this case it does.
- We order the non-terminals as **S, A**.
- Starting from **S (i = 1)**, the inner **j** loop does not execute and we have to eliminate immediate left recursion. But there is no immediate left recursion among **S** productions so nothing happens.

Example on a Grammar (2)

- For the step **`i = 2`**, the inner **`j`** loop, substitutes all occurrences of **`S`** in the A-Productions, producing the following:

`A → Ac | Aad | bd | ε`

- In this case we do have left recursion so proceed to remove it, producing:

`S → Aa | B`
`A → bdA′ | A′`
`A′ → cA′ | adA′ | ε`

Left-Factoring (1)

- Left-factoring is a grammar transformation that produces grammars suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a non-terminal **A**, we defer the decision until we have seen enough input to make the right choice.
- Consider the two productions:

```
stmt → "if" expr "then" stmt "else" stmt
stmt → "if" expr "then" stmt
```

- On seeing the input "IF", it is not clear which production to use to expand the statement.

Left-Factoring (2)

- In general, given two productions:

$\mathbf{A} \rightarrow \alpha\beta_1$
$\mathbf{A} \rightarrow \alpha\beta_2$

- We may rewrite the rules as:

$\mathbf{A} \rightarrow \alpha\mathbf{A'}$
$\mathbf{A'} \rightarrow \beta_1 \mid \beta_2$

- This way we have only one choice for expanding **A**.

Algorithm for Left-Factoring

Input: *Grammar G.*

Output: *An equivalent left-factored grammar.*

Method:

1. For each non-terminal A, find the longest prefix α common to all the alternatives (2 or more).
2. If $\alpha \neq \varepsilon$ (there IS a common prefix)
3. Replace all the productions for A having the prefix α ($A \rightarrow \alpha\beta_1$, $A \rightarrow \alpha\beta_2$, … , $A \rightarrow \alpha\beta_n$) with:

   $\mathbf{A} \rightarrow \alpha\mathbf{A'}$

   $\mathbf{A'} \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

Operator Precedence Parsing of Expressions

- The basis of Operator Precedence Parsing is assigning a priority to each operator in an expression. For example:

```
3 * 4 + 5 = (3 * 4) + 5
```

- Parenthesis may be used to change and visually emphasis precedence:

```
3 * (4 + 5) ≠ 3 * 4 + 5
```

Associativity of Operators

- Associativity for an operator, say ⊗, may be left-associative or right associative:
  - Left-Associative: expression **`x ⊗ y ⊗ z`** is evaluated as **`(x ⊗ y) ⊗ z`**
  - Right-Associative: expression **`x ⊗ y ⊗ z`** is evaluated as **`x ⊗ (y ⊗ z)`**
- For example, given: **`a = 4, b = 5, c = 2`**, the expression **`a - b - c`** may be evaluated as:
  - **`(a - b) - c = (4 - 5) - 2 = -3`**, or
  - **`a - (b - c) = 4 - (5 - 2) = 1`**

Operator Precedence for Parsing Simple Expressions (1)

- Given the following grammar for simple expressions:

```
expr        ::=  <primary> {<op> <expr>};
primary     ::=  <ident>
               | <constant>
               | '-' <primary>
               | '+' <primary>
               | '(' <expr> ')';
```

Parsing the Primaries (1)

```
Function Parse_Primary(token)
   if token = IDENTIFIER then
        // create a leaf with the identifier
        node = IdentLeaf(token)
   else if token = CONSTANT then
        // create a leaf with the constant
        node = ConstantLeaf(token)
   else if token = ADD_OP or token = SUB_OP then
        remember the operator
        lookahead = GetNextToken()
        // parse the primary following the unary op
        primary_node = Parse_Primary(lookahead)
        // combine the unary op and primary in a node
        node = UnaryOpNode(saved operator, primary_node)
```

Parsing the Primaries (2)

```
else if token = OPEN_BRACKET then
     lookahead = GetNextToken()
     // the Parse_Expression function returns the expression
     // node and consumes the next lookahead character
     // which should be the closing bracket
     node = Parse_Expression(lookahead)

     if lookahead <> CLOSE_BRACKET then
             Error - Missing Closing Bracket
     else // read the lookahead
             lookahead = GetNextToken()
     end if
else
     // an invalid character
     Error - Invalid character in primary
end if
```

Parsing the Expression (1)

```
Function Parse_Expression(token, LeftPriority)
   lhs = Parse_Primary(token)
   if Primary Parsed Correctly then
        EndOfExpression = FALSE

        While       (token is an oparator)
                AND (NOT EndOfExpression)

                // see if this operator has a higher priority
                // than the one to its left. The priority
                // of the token to the left is passed as an
                // argument to this function
                if    (Priority of current op)
                    > (LeftPriority) then
                        // this op takes precedence - parse the rhs
                        Remember this operator
                        token = GetNextToken()
```

Parsing the Expression (2)

```
            rhs = Parse_Expression(token,
                                   Priority of current op)

            // combine the lhs to the rhs to act as an
            // lhs for further operators to the right
            lhs = BinaryOpNode(lhs, Current Op, rhs)
      else
            // left op has higher or equal priority
            EndOfExpression = TRUE
      end if
end while

// make expression node in parse tree
node = ExpressionNode(lhs)

return the lookahead at expression node
```

Example (1)

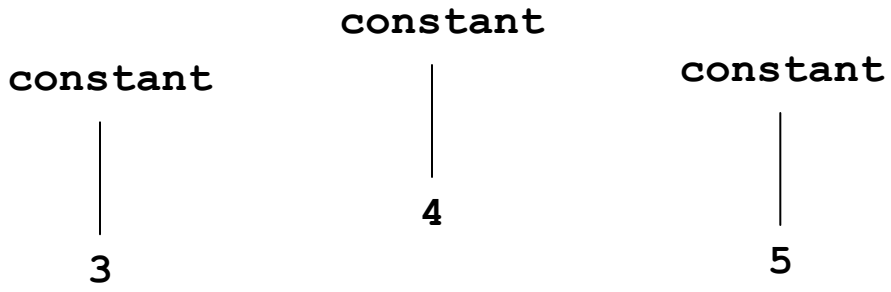
- Consider the parsing of the expression **"3 + 4 * 5 / 6"**
- The first call to Parse_Expression would be **`ParseExpression(3, 0)`**
  - Since there are no operators to the left we pass 0 (the lowest possible priority).
  - The call to **`Parse_Primary`** will create a constant leaf containing **3**.
  - The operator read is a **"+"** which has a higher priority than the one to the left (which actually doesn't exist)
  - An re-invocation to **`Parse_Expression`** is made to parse the rhs.

Example (2)

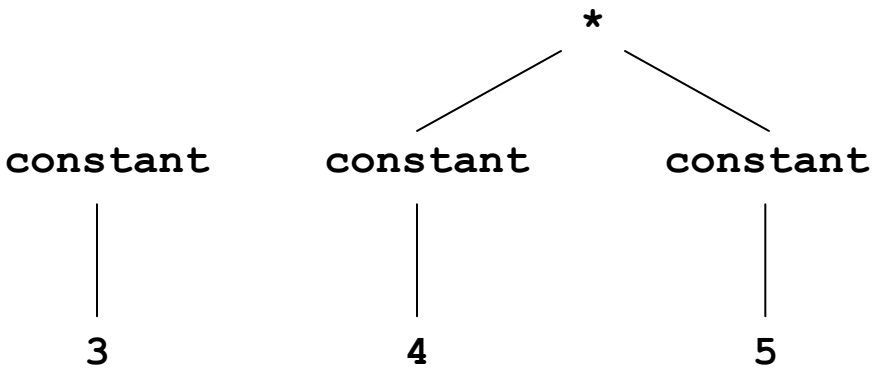
- The call to parse the rhs would be Parse_Expression (4, Priority of “+”).
  - **`Parse_Primary`** will create a constant leaf for **`“4”.`**
  - The operator here is **`“*”`** which has a higher priority than the one on it’s left **`“+”`**
  - Another call to **`Parse_Expression`** is made to parse the rhs.
- The call to parse the rhs would be Parse_Expression(5, Priority of “*”).
  - **`Parse_Primary`** will create a constant leaf for **`“5”.`**
  - The operator here is **`“/”`** which does not have a priority greater than the one to the left **`“*”`** so **`EndOfExpression`** is set to true, the function terminates and control passes to the previous call of **`Parse_Expression(4, Priority of “+”)`**

Example (3)

- So far the tree is:

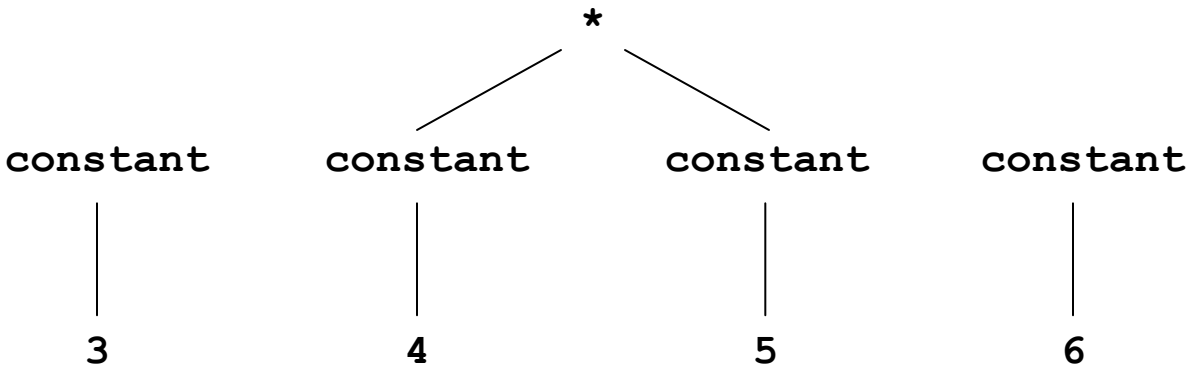


- Now **`Parse_Expression`** has both the LHS and RHS of the multiplication and can join the nodes to get:



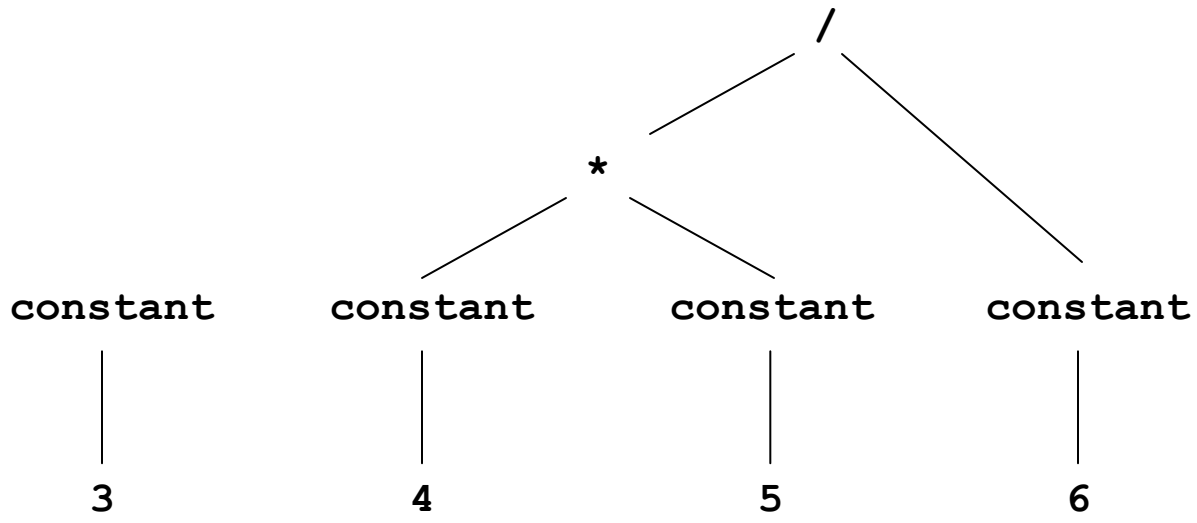
Example (4)

- The multiplication becomes the lhs for the current token "/". This is greater then the priority of the current left operator "+" so a new call to parse is made:
- Parse_Expression(6, Priority of "/")
  - The function will return after creating the constant leaf for "6".



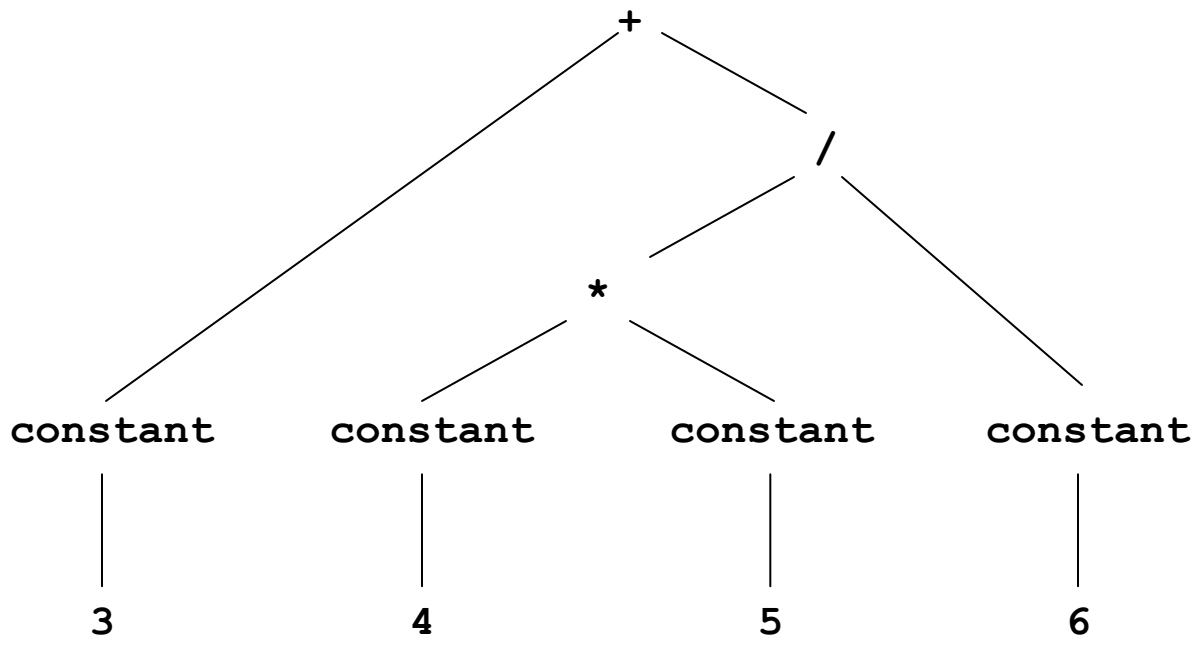
Example (5)

- The function will now return to the previous call that will join the lhs and rhs using the division node:



Example (6)

- Again the function will return to the previous call that will join the lhs and rhs using the addition node:



The Symbol Table

- The purpose of the symbol table table is to record the use of ‘names’ in a program.
- Such names include:
  - Variables, procedure and function names, constants and user defined types.
- The information stored in the symbol table depends on what the names are used for. For example:
  - A variable name requires its type and runtime address.
  - A procedure requires a pointer the list of arguments it takes.
  - A function requires a pointer to the list of arguments it takes and the return type of the function.
  - An argument requires its type and a pointer to the next argument in the list.

Declarations of Variables

- The purpose of variable declarations in programming languages is to create an entry for that variable in the symbol table and associate a type with it.
- Some programming languages (such as earlier versions of BASIC and APL) do not require a declaration and a symbol table entry is made upon their first use.
- When a compiler meets a statement such as `x = 3`, it must verify that:
  - `x` is declared (look for it in the symbol table),
  - `x` is declared as a variable and not, for example, a procedure name,
  - The type of `x` is an integer or floating-point number.

Functions

- When processing a statement such as

  `if f(a, b) then`, the compiler must check:

  - **`f`**, **`a`** and **`b`** are declared,
  - **`f`** takes exactly 2 arguments,
  - **`f`** is a function and returns a boolean value,
  - **`a`** and **`b`** are of the proper types.

Building the Table while Scanning

- When the lexical analyser is scanning the input and meets an identifier, it looks for it in the symbol table:
  - If it **does not find it**, it has to be declared. An entry for that variable is made in the table and its position is returned as the value of the token.
  - If it **does find it**, the position is returned as the value of the token.
- The actual description of the symbol table entry (like its type) is handled by a separate **Object Description Phase**.

Building the Table while Parsing (1)

- A simple lexical analyser does not attempt to process an identifier in anyway. It just returns a token indicating the occurrence of one.
- The actual processing of the identifier is then left to the parser that will deal with it depending on the context in which it has been found. For example, if an identifier is found in a:
  - Declaration Statement, the identifier is looked for in the symbol table. If it is found then the compilers should complain that there is a variable re-declaration. If it does not find it, an entry is made according to the description in the declaration.
  - If the identifier is used in an action statement, a check has to be made to see if it has been declared and that it is used properly (correct number of arguments, no type mismatches, etc…)

Building the Table while Parsing (2)

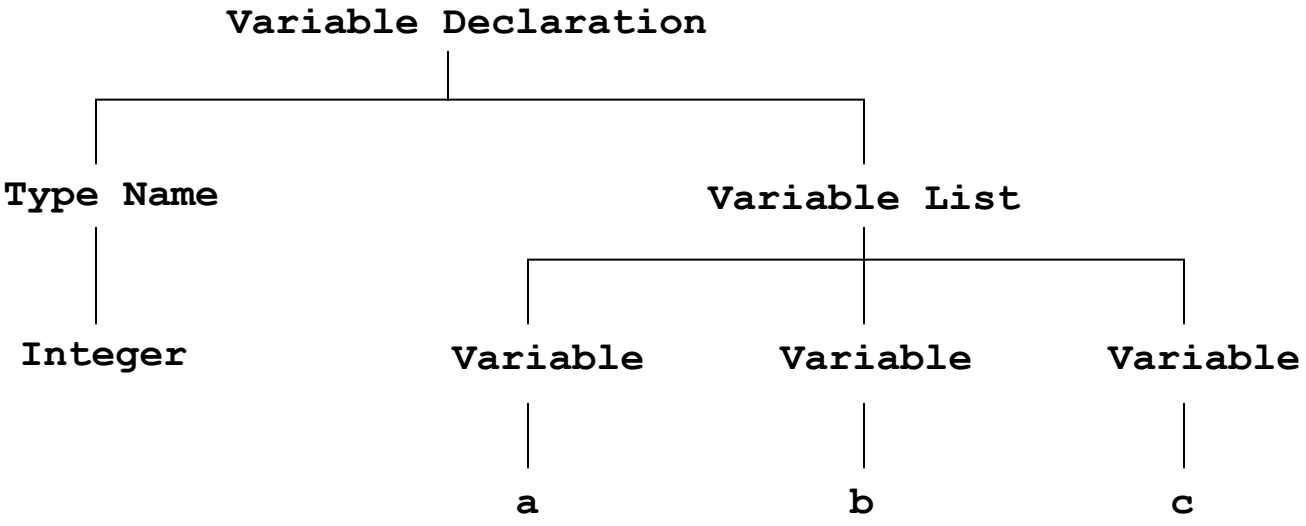
- The method described so far may be implemented in two different ways:
  - The whole parse tree for the variable declaration is built, then declarations in the symbol table and other actions are performed from the tree by the Object Description Phase.
  - Symbol table declarations and other actions are made along the way when parsing.

Symbol Table from the Parse Tree – Option 1

- Suppose we are parsing the declaration:

```
integer a, b, c
```

- The parse tree is passed to an object description phase to analyse it and make the declarations:



Revised Variable Declaration (1) – Option 2

- Note: See slide 67 for original version.
- Consider variable declarations following this format:

```
integer i,j,k
boolean isReady
```

- Recall the grammar:

```
<VarDecl>          ::= <TypeName> <VarNameList>;
<TypeName>         ::= INTEGER|BOOLEAN|REAL;
<VarNameList>      :: <VarName> {“,” <VarName>};
```

Revised Variable Declaration (2)

- Parsing the declaration per se remains the same, so we have no changes so far:

```
Function Parse_VarDecl(TOKEN)
  // Parse the type name
  LOOKAHEAD = Parse_TypeName(TOKEN)

  // Parse the variable name list
  LOOKAHEAD = Parse_VarNameList(LOOKAHEAD)

  Return LOOKAHEAD
End Function
```

Revised Variable Declaration (3)

- Apart from returning the next token, we need to return the type name we just parsed to use later:

```
Function Parse_TypeName(TOKEN)
  If TOKEN is INTEGER_TOKEN then
   Return NextToken, Return Integer Type
  ElseIf TOKEN is BOOLEAN_TOKEN then
   Return NextToken, Return Boolean Type
  ElseIf TOKEN is REAL_TOKEN then
   Return NextToken, Return Real Type
  Else
   Print "Missing Type Name in Declaration"
   Return Error, Return ERROR Type
  End If
End Function
```

Revised Variable Declaration (4)

- Note that when parsing the type name, apart from returning the next lookahead, we also return an indication of which type name we just parsed.
- Parsing the Variable Name List:

```
Function Parse_VarNameList(TOKEN, THE TYPE)
  LOOKAHEAD = Parse_VarName(TOKEN, THE TYPE)
  While LOOKAHEAD = COMMA_TOKEN
   LOOKAHEAD = NextToken
   LOOKAHEAD = Parse_VarName(LOOKAHEAD, THE TYPE)
  End While

  Return LOOKAHEAD
End Function
```

Revised Variable Declaration (4)

- Parsing the variable name:

```
Function Parse_VarName(TOKEN, THE TYPE)
  If TOKEN = IDENT_TOKEN then
   Call procedure VARDECLARE(VARNAME, VARTYPE)
   Note: VARDECLARE is part of the Object
   description phase NOT the parser.
   Return NextToken
  else
   Print "Missing Identifier"
   Return Error
  End If
End Function
```

Final Notes (1)

- The Object Description Phase is a subset of semantic analysis. Ensuring that variables are properly used in action statements is part of another stage of semantic analysis.
- The procedure **VarDeclare**, first looks for the entry of the variable in the symbol table.
  - If the entry is NOT found, a new one is made, recording details (such as the type) of the declaration.
  - If the name is already found, a variable re-declaration might have occurred depending on the **scoping rules** of the language.

Final Notes (2)

- Reuse of the variable declaration is allowed if:
  - All previous uses are no longer in scope.
  - Or, this declaration is made at a lexically lower level than all other active declarations

Final Notes (3)

```
Function VarDeclare(token, VarType)
   Look for a previous occurrence of the Variable

   If no occurrence found then
        Enter details for the variable name and type
   Else
        If      (Use is at a lexically lower level
                than all other active ones) OR
                (Previous uses are not active) then

                Store details in table
        Else
                Re-declaration Error
        End If
   End If
End Function
```

Bottom-Up Parsing

- So far we have seen parsers that try to find a derivation from the starting grammar symbol to the input sting.
- In this section we will be considering the essentially equivalent approach of finding a path from the input sentence to the starting symbol (bottom-up).
- We will be discussing a general technique for bottom-up parsing called shift-reduce parsing and an implementation of it called LR parsing.
- This method is used in automatic parser generators such as BISON, which we will cover in the next lessons.

Notes on Shift-Reduce Parsing

- Bottom-Up parsing can be thought of **reducing** the input string to the starting symbol of the grammar.
- Each reduction step involves a sequence of symbols from the input being replaced by a left hand side non-terminal according to the grammar rules.
- Just as in producing a Top-Down derivation there may be several non-terminals that may be expanded at any one step, in bottom-up parsing, there may be several sequences of symbols in the input string that may be reduced in a step.

LL and LR Parsing

- It is important to choose a parsing methodology to apply to each derivation or reduction and apply it consistently.
- In in top-down parsing we always expand the leftmost non-terminal in a sentential form then we obtain a **leftmost** derivation. If we always expand the rightmost non-terminal then the derivation is **rightmost**.
- The rule that we will apply to bottom-up parsing is to always reduce the sequence of symbols which would trace a rightmost derivation in reverse. So, the input is scanned from left to right (hence LR).
- LL parsers scan the input from left to right to produce a leftmost derivation.

Example (1)

- Consider the following grammar

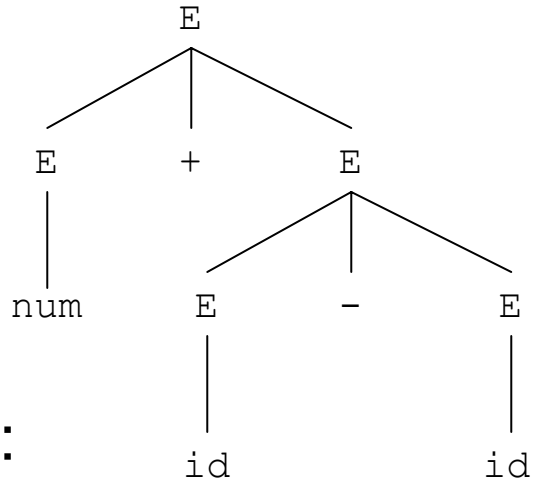
```
E ::= E + E
E ::= E - E
E ::= id
E ::= num
```

- And the input string '`1 + x - y`'. A leftmost derivation would be:

```
E → E + E
  → num + E
  → num + E - E
  → num + id - E
  → num + id - id
```

Example (2)

- The parse tree would be:



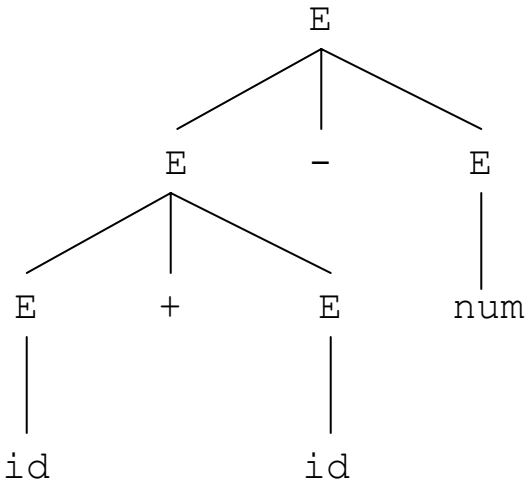
- A rightmost derivation would be:

E → E + E
→ E + E – E
→ E + E – id
→ E + id – id
→ num + id – id

- Which gives the same parse tree.

Example (3)

One should note however that the grammar is ambiguous. We could have the following derivations:



Leftmost	Rightmost
E → E – E	E → E – E
→ E + E – E	→ E – id
→ num + E – E	→ E + E – id
→ num + id – E	→ E + id – id
→ num + id – id	→ num + id – id

Rightmost Derivations in Reverse (1)

- The first rightmost derivation we had was:

```
E → E + E
  → E + E – E
  → E + E – id
  → E + id – id
  → num + id – id
```

- If applied in reverse we get a bottom-up parse:

```
num + id – id → E + id – id
              → E + E – id
              → E + E – E
              → E + E
              → E
```

Rightmost Derivations in Reverse (2)

- The second rightmost derivation we had was:

```
E → E – E
  → E – id
  → E + E – id
  → E + id – id
  → num + id – id
```

- If applied in reverse we get a bottom-up parse:

```
num + id – id → E + id – id
              → E + E – id
              → E – id
              → E – E
              → E
```

Handles (1)

- The term **input string** will be used to refer to any sentential form in the reduction of a sentence to the starting symbol.
- The term **handle** is a sequence of symbols in the input string which, if replaced by a matching left hand side non-terminal, leads to the tracing out of the reversed rightmost derivation of the original sentence.
- Consider the example we had before in reducing `num + id - id` to `E`.

Handles (2)

Right Sentential Form	Handle	Reducing Production
num + id1 – id2	num	E → num
E + id1 – id2	id1	E → id
E + E – id2	id2	E → id
E + E – E	E – E	E → E – E
E + E	E + E	E → E + E
E		

Handles (3)

Right Sentential Form	Handle	Reducing Production
num + id1 – id2	num	E → num
E + id1 – id2	Id1	E → id
E + E – id2	E + E	E → E + E
E – id2	id2	E → id
E – E	E – E	E → E – E
E		

Handles (4)

- Note that upon having reached the input sting '`E + E - id`', there are two possible handles '`id`' or '`E + E`'. This reflects the ambiguity in the grammar. If a grammar is unambiguous, then for any input string there would be only one handle for any stage in the reduction.
- An issue in designing a bottom-up parser is to
  - Decide how to locate handles in the input string.
  - How to choose which left hand side to replace with assuming that there may be more than one left hand side for a handle.

Stack Implementation of Shift Reduce Parsing (1)

- Shift-reduce parsers are usually implemented using a stack to hold grammar symbols and an input buffer to hold the string X to be parsed.
- We shall use the dollar symbol, $, to denote the bottom of the stack and the end of the input string.
- Initially we would have:

Stack	Input String
$	X$

Stack Implementation of Shift Reduce Parsing (2)

- The parse works by shifting symbols from the input string to the stack until a handle appears on the top of the stack.
- The parser reduces the handle on top of the stack to the left hand side of the appropriate production.
- The cycle is repeated until we find an error or the stack consists of the starting symbol and the input is empty.
- At this point, the parser stops and reports success.

Stack	Input String
Y$	$

Example (1)

- Consider the following unambiguous grammar…

```
E → E + T
E → E – T
E → T
T → id
T → num
```

- …to parse ‘`num + id + id`’

Example (2)

Stack	Input	Action
$	num + id – id $	Shift num
$ num	+ id – id $	Reduce T → num
$ T	+ id – id $	Reduce E → T
$ E	+ id – id $	Shift +
$ E +	id – id $	Shift id
$ E + id	– id $	Reduce T → id
$ E + T	– id $	Reduce E → E + T
$ E	– id $	Shift –
$ E –	id $	Shift id
$ E – id	$	Reduce T → id
$ E – T	$	Reduce E → E – T
$ E	$	Accept

Parsing Actions

- Shift:
  - In a shift operation, the next input symbol is put on the top of the stack.
- Reduce:
  - The handle which is on top of the stack is replaced by the appropriate non-terminal symbol.
- Accept:
  - The parser accepts when all the input symbols are consumed and there is the sentence symbol on the stack.
- Error:
  - The parser encounters an error an calls any error recovery routine.

Shift-Reduce Conflicts (1)

- There are grammars that cannot be parsed by a shift reduce parser. In such cases, the parser can get into a state in which it cannot decide whether to shift or reduce.
- Ambiguous grammars are of such a type because there may be more than one handle at a time under certain circumstances.
- Consider the dangling else grammar:

  `S → if E then S | if E then S else S`

Shift-Reduce Conflicts (2)

- The shift-reduce parser may find itself in the following situation:

Stack	Input String
`$ if E then if E then S`	`Else S $`

- At this point the parser wouldn't know whether to shift the else onto the stack or reduce the first production for s to get:

Stack	Input String
`$ if E then S`	`Else S $`

Good News

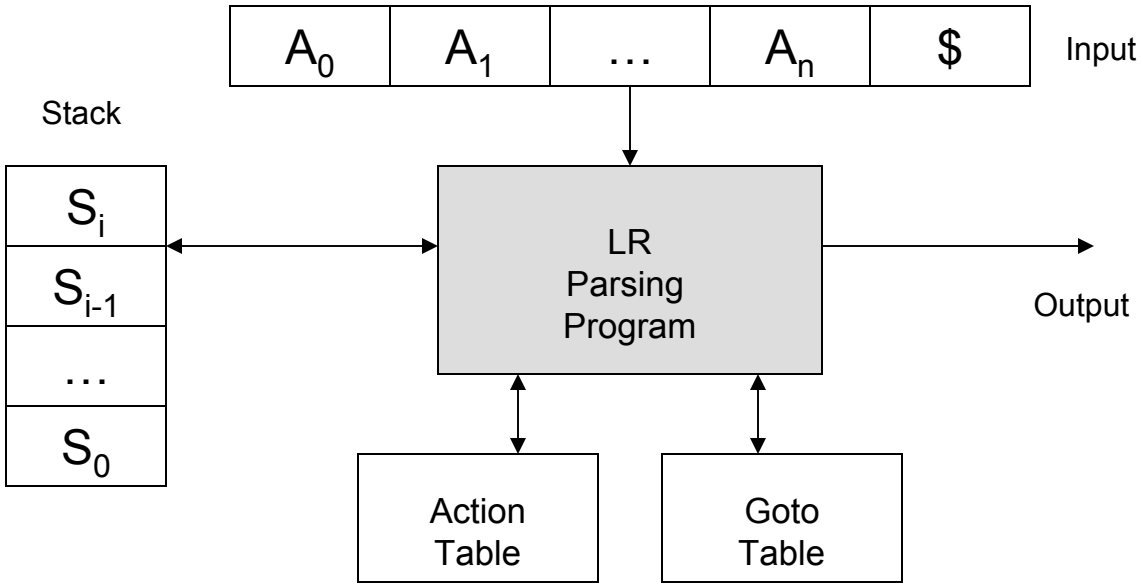
- Shift-reduce parsers may be easily modified to handle such grammars in a consistent way. For example, such shift-reduce conflicts may be resolved by forcing the parser to shift.
- Shift-reduce conflicts are not very common and are often an indication that there is a problem in the definition of the language.

LR(k) Parsing

- LR Parsing is an efficient bottom-up parsing technique that can be used to parse a large class of context-free grammars.
- LR means that the input is scanned from left-to-right building a rightmost derivation in reverse.
- k represents the number of lookahead symbols required to make a parsing decision.
- If k is omitted it is assumed to be 1. Many grammars in compiling fall into the LR(1) class of parsers.
- The main advantage of LR parsers is that they be made to recognise virtually any language for which a context-free grammar exists.
- The main drawback however is that they tend to be very complicated to code by hand, however may generators exist that take a context-free grammar as input and produce a parser for it.

Design (1)

- An LR shift-reduce parser consists of an input, an output, a stack, a parsing program and two parsing tables (action and goto):



Design (2)

- The stack is used to store parsing states.
- The state on the top of the stack combined with the next input token are used by the parsing program to deduce whether it has a handle to reduce or whether it should shift a new state on top of the stack and read the next input token.
- Each entry in the action table contains the four actions for any combination of top stack symbol and next token Si, Aj:
  - Shift,
  - Reduce,
  - Accept,
  - Error.

Design (3)

- The goto table is used whenever the action is a reduction. After a reduction X → α, the states corresponding to the handle α are popped from the stack to expose the new topmost state s' and the entry for goto[s',X] becomes the new state on top.

Algorithm (1)

- The parser starts with an initial state s_0 on the stack. At some point through a parse the stack will contain $s_0 s_1 s_2 \ldots s_i$.
- Given the next input token a, the parser will proceed as follows:
  - If action[s_i,a] = shift s_{i+1}, the new state is put on top of the stack to become: $s_0 s_1 s_2 \ldots s_i s_{i+1}$, and the new token is read.
  - If action[s_i,a] = reduce $Y \rightarrow X_1 \ldots X_k$, then the k states $s_{i-k+1} \ldots s_i$ are popped off the stack leaving s_{i-k} on top. Now, goto[s_{i-k},Y] is consulted to find a new topmost state s_{i-k+1} which is put on top of the stack to become: $s_0 s_1 s_2 \ldots s_{i-k} s_{i-k+1}$.

Algorithm (2)

- If action [s_i,a] = accept, then the parsing is complete – the whole input tokens have been consumed and reduced to the sentence symbol.
- If action [s_i,a] = error, then a syntax error has been detected.

Parsing Program (1)

```
Set pointer ip to point to the input string
Repeat forever
  Let s = state on top of stack
  Let a = symbol pointed to by ip

  if action[s,a] = shift s' then
    push s' on top of stack
    increment ip to next symbol

  else if action[s,a] = reduce A→B then
    for i = 1 to length(B)
      pop state from stack

    Let s' be the new state on top of stack
    Let s'' = goto[s',A]
    Push s'' on top of stack
```

Parsing Program (2)

```
  else if action[s,a] = accept then
    exit from infinite loop

  else if action[s,a] = error then
    report error
End Repeat
```

LR Parsing Example (1)

- Consider the expression `id * id + id`.
- Grammar:

```
1)  E → E + T
2)  E → T
3)  T → T * F
4)  T → F
5)  F → (E)
6)  F → id
```

LR Parsing Example (2)

State	Action Table						Goto Table		
	id	+	*	(	)	EOF	E	T	F
0	S5			S4			1	2	3
1		S6				Acpt			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	R7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

LR Parsing Example (3)

- Notes:
  - s_n means shift state n and read the new token.
  - r_k means reduce by the production k.
  - Encountering blank entries in the tables signify an error.
  - The value goto[s,a] for a TERMINAL a, is found in the action field action[s,a]. The goto table, therefore contains values for goto[s,a] where a is a NON-TERMINAL.

LR Parsing Example (4)

	Stack	Next Token	Action
1	S_0	id	Shift S_5
2	S_0S_5	*	Reduce F → id
3	S_0S_3	*	Reduce T → F
4	S_0S_2	*	Shift S_7
5	$S_0S_2S_7$	id	Shift S_5
6	$S_0S_2S_7S_5$	+	Reduce F → id
7	$S_0S_2S_7S_{10}$	+	Reduce T → T * F
8	S_0S_2	+	Reduce E → T
9	S_0S_1	+	Shift S_6
10	$S_0S_1S_6$	id	Shift S_5
11	$S_0S_1S_6S_5$	EOF	Reduce F → id
12	$S_0S_1S_6S_3$	EOF	Reduce T → F
13	$S_0S_1S_6S_9$	EOF	Reduce E → E + T
14	S_0S_1	EOF	Accept

LR Parsing Example (5)

- At the beginning the parser is in state `0` with `id` as the first input token.
- Therefore `action[0,id]` is taken giving the state s_5 which is pushed on the stack and the new input token is read.
- `'*'` is now the input symbol and the `action[5,*]` is to reduce by rule `6`. One state is popped off (one state on right hand side) exposing state `0`. The value for `goto[0,F]` is `3` meaning that state `3` must be popped onto the stack.
- `'*'` is still the input symbol and `action[3,*]` is to reduce by rule `4`. One state is popped off (one state on right hand side) exposing state `0`. The value for `goto[0,T]` is `2` meaning that state `2` must be popped onto the stack.
- And so on…

Constructing The Parsing Tables

- There are 3 widely used LR parsing techniques:
  - Canonical LR(k) or LR(k) is the most general form of LR parsing methods and is the most powerful. Such parsers usually have many thousands of states for a programming languages and are VERY difficult to hand code.
  - Simple LR(k) or SLR(k) is a variant of LR(k) parsing and usually involves a few hundred states. SLR parsers are the weakest in terms of grammars it can handle but serves as a good starting point to other LR parsing methods.
  - Lookahead LR(k) or LALR(k) is somewhat in the middle in terms of the grammars it can handle. LALR parsers have the same number of states as the equivalent SLR parser but are more difficult to construct. Popular parser generators use this technique to automate parser generation.

FLEX

- FLEX is a popular program that generates lexical analysers.
- FLEX accepts as an input a description of the scanner it has to generate and produces a C source file called `'lexyy.c'` containing the scanning code.
- By convention, FLEX input files have the extension `'.l'`.

The FLEX Input File

- The general format of a FLEX source file is:

```
Definitions
%%
Rules
%%
User Subroutines
```

- The definitions and user subroutines sections are optional as is the second set of `%%` delimiters. Note that the first set of delimiters is required to separate the definitions section from the rules.
- The absolute minimum FLEX program is:

```
%%
```

- Which copies the input program to the output unchanged.

FLEX Definitions (1)

- The definition sections contains declaration of language constructs to simplify the scanner specification.
- These declarations have the form:

```
name definition
```

- Where name is any alpha-numeric word starting with an underscore of a letter.
- For example:
  - `Digit [0-9]`
  - `Ident [a-z][a-z0-9]*`
- Where Digit defines a regular expression that recognises a simple one-character digit and Ident recognises a word starting with a letter followed by zero or more occurences of a letter or digit.

FLEX Definitions (2)

- Thus, a subsequent call to:

  `{digit}* ”,” {digit}+`

- Is identical to

  `([0-9])* ”.” ([0-9])+`

- In the definitions section, any indented text or text enclosed within `%{` and `%}` is copied to the output as it is with the `%{` and `%}` removed.
- The text lines within are usually:
  - Compiler directives such as `#include`’s or `#define`’s.
  - Declarations of variables that are used by other sections of the FLEX input file.

FLEX Rules

- The rules section of a FLEX program contain a series of `pattern action` statements.
- For example:

```
Integer puts("I found an integer")
```

- Would print a message each time an Integer (defined in the definitions section) is found.
- Any indented or '`%{...%}`' code appearing before the first rule in this section is local to the main scanning routine generated by FLEX and is executed each time the routine is called.

FLEX User Subroutines

- The user subroutines section is copied exactly to the output source produced by FLEX.
- When building FLEX scanners that are not interfaced by external programs the C `main` function is defined and programmed here.

Regular Expressions in FLEX (1)

`x`	Matches the character x.
`.`	Any character except the newline.
`[xyz]`	A ‘character class’ – in this case it matches an ‘x’ a ‘y’ or a ‘z’
`[abj-oZ]`	A character class with a range in it – matches an ‘a’, a ‘b’, any letter from ‘j’ to ‘o’ or ‘Z’.
`[^A-Z]`	A negated character class.
`[^A-Z\n]`	A negated character class with an escape character (newline).
`r*`	Zero or more r’s where r is a regular expression.
`r+`	One or more r’s.
`r?`	An optional r (zero or one)
`r{2,5}`	Anything between 2 and 5 r’s.
`r{2,}`	2 or more r’s.

Regular Expressions in FLEX (2)

`r{4}`	Exactly 4 r's.
`{name}`	The expansion of a name definition.
`"[xyz]"`	The literal '[xyz]'
`\X`	If X is an a,b,f,n,r,t or v, the ANSI C interpretation of \X otherwise the literal X – for example \"
`\123`	A character with the octal value of 123.
`\x2a`	A character with the hexadecimal value 2a.
`(r)`	Match an r – parenthesis are used to emphasis precedence.
`rs`	Concatenation of regular expression r and s.
`r|s`	Either an r or an s.
`^r`	An r but only ath the beginning of a line.
`r$`	An r but only at the end of a line – equivalent to r\n.
`<<EOF>>`	The end of file.

Regular Expressions in FLEX (3)

- If there is more than one rule matching the input, the one matching the most text characters is chosen. If the matches have the same length, the file listed first is chosen.
- Once a match is made, the text corresponding the match is put in a special character pointer (C string) variable called `yytext` (`char *yytext`) and its length is in `yyleng`.
  - `Integer printf("I found an integer %s.", yytext);`

Actions (1)

- Each pattern in a rule has a corresponding action that may be any arbitrary C statement.
- The pattern ends at the first non escaped whitespace character. The rest of the line is the action statement.
- If the action is left empty, the token found is discarded.
- The following FLEX program deletes all occurrences of the word ‘username’ from the input an keeps the rest:

```
%%
"username"
```

- The following program compreses multiple spaces and tabs into one space charater and removes trailing spaces too:

```
[ \t]+   putchar(' ');
[ \t]+$  /* ignore trailing blanks */
```

Actions (2)

- An action consisting of only the vertical bar '|' means "the same action as the one for the next rule. If the action contains a '{', then the action spans until the next balancing '}'. For example:

```
IF  |
if  {
                    puts{"Keyword IF found.");
                    return IFWORD;
            }
```

- Actions contain arbitrary C code, including return statements to return values to whatever external routine called `yylex()` – the token parser.
- Each time `yylex()` is called, it continues processing from where it last left off until it reaches an `EOF` or meets a return statement.
- Once `yylex()` reaches the end of file, however, any subsequent call to `yylex()` will immediately return unless `yyrestart()` is called.
- Note any actions are not allowed to modify `yytext` or `yyleng`.

Special Routines and Directives (1)

- `ECHO`: copies `yytext` to the scanners output.
- `yymore()`: tells the scanner that the next time it matches a rule, the corresponding token should be appended onto the current value of `yytext` rather than replacing it. For example:

```
%%
a-          ECHO; yymore();
b           ECHO;
```

- The first '`a-`' is matched and echoed to the output. Then '`b`' is matched by the previous '`a-`' is still in `yytext` so the echo for '`b`' will include the previous '`a-`'s

Special Routines and Directives (2)

- `yyless(n)`: redirects all but the first `n` characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match. `yytext` and `yyleng` are adjusted appropriately. Note that a call to `yyless(0)` will cause the entire input string to be scanned again and would result in an infinite loop unless care is taken.
- unput(c): puts the character '`c`' back onto the input stream which will then be the next character scanned. For example the following action will take the current token and cause it to be rescanned enclosed in parenthesis:

```
{
   int i;
   unput(')');
   for (i = yyleng -1; i > 0; --i)
     unput( yytext[i] );
   unput('(');
}
```

- Note that all characters are pushed to the BEGINNING of the input string so the original characters are put in reverse.

Special Routines and Directives (3)

- `input()`: reads the next character from the input stream. (or `yyinput()` if used with C++)
- `yyterminate()`: can be used instead of a return statement. It aborts the action returning 0. subsequent calls to `yylex()` immediately return unless `yyrestart()` is called. (usually called on encountering the EOF)
- `yyrestart()`: tells FLEX to start scanning from a new (maybe the same) input file. Takes a single `file *` pointer.

The Generated Scanner (1)

- Whenever `yylex()` is called, it scans tokens from a global input file denoted by `yyin` which by default points to standard input unless specified using the C function `fopen` (file open).
- For example, to open `example.txt` for reading in text mode:

```
yyin = fopen("example.txt","r");
```

- `yylex()` continues reading from `yyin` until it reaches EOF. In this case the function will return immediately unless yyrestart() is called and yyin is set to point to a new file.

The Generated Scanner (2)

- Likewise, the scanner produces output to `yyout` which, again, by default, points to standard output. As with `yyin`, `yyout` can be changed by assigning it another `FILE` pointer:
  - `yyout = fopen("output.txt", "w");`

Interfacing with Parser Generators

- One of the main uses of FLEX is interfacing it with an external parser generator like Bison. Bison parsers expect to call a function called `yylex()` to find the next input token.
- `yylex()`, is expected to return the type of the token found (implemented as a constant, maybe) and putting any associated value in `yylval`.
- To use Bison in association with FLEX, it is called with the `-d` option to instruct is to generate an header file containing all the token definitions. This header is then used in FLEX. To include the header:

```
%{
#include "generated_header.h"
%}
%%
[0-9]+    yylval = atoi(yytext); return NUM;
```

BISON

- BISON takes an input grammar file and produces a C program that parses the language described by that grammar.
- Tokens are read from the lexical analyser function `yylex()` which can be coded manually or generated automatically using FLEX.
- The BISON output file (C program) defines a function called `yyparse()` – the implementation of the grammar.
- The parser generated by BISON expects a user-implemented error reporting function `yyerror()`. And the `main()` C function.

BISON Grammar Files

- A BISON grammar file contains four sections separated by delimiters:

```
%{
C Declaration
%}

Bison Declarations

%%
Grammar Rules

%%
Additional C Code
```

The C Declarations Section

- This section contains global definitions, constants, variables, #include's, #define's and functions that will be used in the actions of the grammar rules.
- The contents of this section are copied to the very beginning of the output parser file so that they precede the `yyparse()` function.
- If no C declarations are used, the `%{` and `%}` delimiters may be omitted.
- By now you should have noticed that both BISON and FLEX have a lot of variable and function definitions starting with yy. It is a good idea NOT to name any variables or functions of your own starting with yy too.

Other Sections

- Bison Declarations
  - This section contains declarations of terminal and non-terminal symbols used in the language being described, as well as definitions of operator precedence and the data types of semantic values of various symbols.
- Grammar Rules Section
  - This section contains the grammars production rules, which define how a non-terminal is constructed from its parts. There must always be at least one grammar rule in a BISON file.
- Additional C Code
  - Like the C Declarations section, the contents of this section contains C code that is copied exactly to the output. This section is copied to the END of the output file. It is a convenient way to put anything required AFTER the `yyparse()` function such as `main()`.

Symbols – Terminal and Non-Terminal (1)

- A terminal symbol represents lexical analyser tokens. These tokens are represented by numeric constants, any `yylex()` returns a token type code to indicate what token type has been read.
- There are 2 ways of writing terminal symbols in the grammar:
  - Single Characters: A single character token type such as + or * does not need to be declared. It can be used directly in the rules section by enclosing it in single quotes.
  - Multi-Character Tokens: These are represented by a declared name using a `%token` declaration. By convention names are written in upper case. For example the words “Begin” and “End” might be declared as:

```
%token  BEGIN
%token  END
```

-or-

```
%token  BEGIN END
```

Symbols – Terminal and Non-Terminal (2)

- Internally, each token is represented by an integer, starting from 257. 0 to 255 are used to represent ASCII characters and 256 is used to represent the error token. When using BISON the programmer is not generally concerned about these values but these token values must be known when implementing these tokens in FLEX. Using the '–d' option when running BISON will automatically generate an include file containing the definitions for these tokens:

```
…
#define BEGIN 257
#define END 258
…
```

Symbols – Terminal and Non-Terminal (3)

- Non-terminals are declared in exactly the same way, but their names are in lowercase by convention.

BISON Grammar Rules (1)

- A BISON grammar rule has the form:

```
result      : components...
            ;
```

- Where result is the non-terminal symbol that the rule describes (LHS) and the components are the various terminal and non-terminal symbols that put together this rule (RHS).
- For example:

```
exp                 : exp '+' exp
                    ;
if_statement        : IF exp THEN
                    ;
```

BISON Grammar Rules (2)

- Multiple rules for the same result can be written separately:

```
exp     : exp '+' exp;
exp     : exp '-' exp;
```

- Or together, separated by the vertical bar:

```
exp     : exp '+' exp
        | exp '-' exp
        ;
```

- If the components section is left empty, it means that result can match the empty string.

BISON Grammar Rules (3)

- Here is how to define a comma separated sequence of zero or more `exp` groupings:

```
expseq          : /* empty */
                | expseq1
                ;
expseq1         : exp
                | expseq1 ‘,’ exp
                ;
```

- It is convention to write a comment `/* empty */` in each rule with no component.
- Within components actions consisting of C statements may be included:

```
exp : exp ‘+’ exp { printf(“Addition Expression”);}
    | exp ‘-’ exp { printf(“Subtraction Expression”);}
    ;
```

Recursive Rules (1)

- A rule is called recursive when its result also appears also on the right-hand side. Nearly all BISON grammars need to use recursion, because it is the only way to define a sequence (zero-or-more, one-or-more) of ‘somethings’.
- Consider the left and right recursive definitions of a comma-separated sequence of one or more expressions:

```
expseqleft: exp
                | expseqleft ‘,’ exp
                ;
expseqright     : exp
                | exp ‘,’ expseqright
                ;
```

Recursive Rules (2)

- Any kind of sequence may be defined using either left or right recursion, but one should always use left recursion, because it can parse a sequence of any number of elements with bounded stack space.
- Indirect or mutual recursion occurs when the result of the rule does not appear directly on the right hand side, but does appear in rules for other non-terminals which do appear on its right hand side. For example:

```
expr            : primary
                | primary '+' primary
                ;

primary         : constant
                | '(' expr ')'
                ;
```

Defining Semantics (1)

- The grammar rules for a language determine only its syntax. The semantics are determined by the semantic ‘meaning’ associated with various tokens and the actions taken when these tokens are recognised.
- A formal grammar selects tokens only by their classifications: for example, if a rule mentions the terminal symbol `integer constant', it means that *any* integer constant is grammatically valid in that position. The precise value of the constant is irrelevant to how to parse the input: if `x+4' is grammatical then `x+1' or `x+3989' is equally grammatical.

Defining Semantics (2)

- Semantic values have all the rest of the information about the meaning of a token, such as the value of an integer, or the name of an identifier. (A token such as ',' which is just punctuation doesn't need to have any semantic value.)
- For example, an input token might be classified as token type INTEGER and have the semantic value 4. Another input token might have the same token type INTEGER but value 3989. When a grammar rule says that INTEGER is allowed, either of these tokens is acceptable because each is an INTEGER. When the parser accepts the token, it keeps track of the token's semantic value.
- Each grouping can also have a semantic value as well as its non-terminal symbol. For example, in a calculator, an expression typically has a semantic value that is a number. In a compiler for a programming language, an expression typically has a semantic value that is a tree structure describing the meaning of the expression.

Defining Semantics (3)

- Most of the time, the purpose of an action is to compute the semantic value of the whole construct from the semantic values of its parts. For example, suppose we have a rule which says an expression can be the sum of two expressions. When the parser recognizes such a sum, each of the sub-expressions has a semantic value which describes how it was built up. The action for this rule should create a similar sort of value for the newly recognized larger expression.
- The C code in an action can refer to the semantic values of the components matched by the rule with the construct `$`*n*, which stands for the value of the *n*th component. The semantic value for the grouping being constructed is `$$`. (Bison translates both of these constructs into array element references when it copies the actions into the parser file.)
- Here is a typical example:

```
exp:        ...
          | exp '+' exp { $$ = $1 + $3; }
```

Defining Semantics (4)

- If you don't specify an action for a rule, Bison supplies a default: `$$ = $1`.
- Every terminal and non-terminal defined in the grammar is given a type. Bison's default is to use the `int` type for all semantic values. Clearly, this can be overridden.

BISON Declarations

- This section defines all the symbols used in formulating the grammar and the data types of semantic values.
- All token types except for single character tokens (such as +, which are enclosed in single quotes) must be declared.
- Non-terminal symbols must be declared if you need to specify which data type to use for the semantic value.

The Sentence Symbol

- The sentence symbol of the grammar is, by default, the first non-terminal defined at the start of the rules section. An alternative start symbol may be specified using the `%start` statement.

  For example, if the starting symbol in your language is `program`, you may specify this as:

  ```
  %start program
  ```

Token Types

- The basic way to specify a token is using the `%token` statement:

```
%token begin
%token end
```

- One can explicitly specify a numeric code to each token type:

```
%token begin      300
%token end        301
```

- But, in general, it is better to let BISON choose the numeric code itself.

Types of Semantic Values (1)

- The BISON `%union` declaration is used to specify the collection of all possible data types for all the semantic values:

```
%union
{
  double val;
  char * str;
}
```

- This means that we defined two types – val (based on the double type) and str (a C string).

Types of Semantic Values (2)

- In certain cases, token declarations (`%token …`) should be assigned a type. For example, if the token `NUM` must be associated to the semantic type `double`, then the token declaration should be modified as:

```
%token <val> NUM
```

- We previously mentioned that in some cases, non-terminals could be associated with a semantic type. In this case the non-terminal declaration is mandatory. Suppose the EXPR and PRIMARY non-terminals are associated a double type:

```
%type <val> EXPR PRIMARY
```

Associativity

- If one wishes to declare a token and specify its associativity the `%left`, `%right` and `%nonassoc` statements are used.
- If '+' is declared to be left associative:

```
%left '+'
```

- The same reasoning goes for right associativity. A token may be non-associative. Say, '+' should be declared with no associative information. We get:

```
%nonassoc '+'
```

- But keep in mind that statements like `'a+b+c'` will be considered as a syntax error (we have more than one operator but don't have associativity information)

Precedence

- All tokens declared together have the same precedence.
- When tokens are declared separately, the one declared later has the highest precedence.

```
%left '+' '-'
%left '*'
```

Type Checking (1)

- There are 2 classes of ‘checking’ that are made when during the lifetime of a program, namely, static and dynamic checking. Dynamic checking occurs during the execution (runtime) of a target program. Static checking is made at compile time.
- Examples of static checks include:
  - Type checks: a compiler should produce an error id an operator is applied to an incompatible operand.
  - Flow control checks: statements that effect the flow of a program must have a ‘place’ were to redirect the flow. For example, the C `break` statement causes the control to leave the enclosing `while`, `for` or `switch` statement. An error occurs if there is no such enclosing statement.
  - Uniqueness checks: there are situations were an object must be defined only once such as a variable declaration.
  - Name-related checks: sometimes, a name must appear two or more times (`for` **`i`** `= a to b … next` **`i`**). The compiler must check that the same name is used in both places.

Type Checking (2)

- A type checker verifies that the type of a construct ‘fits’ into its current context. For example the Pascal `mod` operator requires integer operands, so the compiler must ensure that this is so.
- A symbol that can represent different operations in differing context is said to be ‘overloaded’.
- In principal any check can be made dynamically, if the target code contains enough type information.
- A strongly typed language is one that guarantees that if the compiler accepted the input, it will run without type errors.
- In practice there are some check that can be made only dynamically. For example if we declare an array table: `array[0..255] of char;` and try to reference `table[i]`, the compiler cannot guarantee during execution that the value `i` will lie in the `0..255` range.

A Simple Type Checker (1)

- We will specify a small language in which every identifier must be declared before being used.
- The following grammar generates programs starting from the starting symbol P consisting of a sequence of declarations D followed by a single expression E.

```
P → D ; E
D → D ; D | id : T
T → char | integer |
                  array[ num ] of T | ^T
E → literal | num | id | E mod E |
                  E [ E ] | E^
```

A Simple Type Checker (2)

- A program that can be generated from the grammar is:

```
key: integer;
key mod 1999
```

- Notes:
  - The basic types in the language are char and integer.
  - We assume all array indices start from 1 so `array[256]` is the equivalent of `array[1..256]`.
  - The prefix `^` operator is the pointer type.
  - In the translation scheme we will use, the action associated with the production `D` → **`id :`** `T`, will save the type information for the identifier in the symbol table.
  - Since in the grammar, `D` appears before `E` in `P → D ; E` it is guaranteed that all the types of identifiers will be known before the expression is checked.

Type Checking Expressions (1)

- The following rules, synthesize the the type of an expression :

E → literal	E.Type ::= char
E → num	E.Type ::= integer
E → id	E.Type ::= lookup(id) *Where **lookup** searches for **id** in the symbol table and returns the type of the declared identifier.*
E → E_1 mod E_2	if E_1.Type AND E_2.Type = integer E.Type = integer else E.Type = type_error

Type Checking Expressions (2)

$E \rightarrow E_1[E_2]$	if E_2.Type = integer and E_1.Type = array(s,t) E.Type = t else E.Type = type_error *In **array, s** is the size and **t** is the type.*
$E \rightarrow E_1\hat{}$	if E_1.Type = pointer(t) E.Type = t Else E.Type = type_error *Where **t** in **pointer(t)** is the type of the pointer.*

Type Checking Statements (1)

- Certain language constructs like statements don't have values *per se* so don't have types associated with them. In this case a special basic type *void* can be assigned to them. If an error is detected the *type_error* type is returned.
- We will be considering `assignment`, `while` and `if` statements here.
- Sequences of statements are separated by semicolons.

Type Checking Statements (2)

S → **id** := E	if id.Type = E.Type
	  S.Type = void
	else
	  S.Type = type_error

S → **if** E **then** S_1	if E.Type = boolean
	  S.Type = S_1.Type
	else
	  S.Type = type_error

S → **while** E **do** S_1	*-same as above-*

S → S_1 ; S_2	if S_1.Type = void and S_2.Type = void
	  S.Type = void
	else
	  S.Type = type_error

Runtime Support (1)

- Before discussing code generation, we will examine the relationship between the text of the source program to the actions that have to occur at runtime to implement it.
- The execution of every procedure is referred to as an activation of that procedure.
- If procedures are nested or recursive multiple activations may exist at any one point.
- Let us assume that a program is made up of procedures such as in Pascal.
- In its simplest form a procedure is the relationship between and identifier and a statement, where the identifier is the procedure name and the statement(s) is the procedure body.
- Procedures that return a value are called functions in many programming languages.

Runtime Support (2)

- A complete program will also be treated as a procedure (think Pascal).
- When a procedure appears in an action statement, we say that the procedure has been *called*.
- A procedure may be also called within an expression.
- Some identifiers within a procedure definition are treated special and are called the *formal parameters* of the procedure (also called arguments).
- When a procedure is called, *actual parameters* are substituted for the formal ones.

Activation Trees (1)

- Assumptions on flow control:
  - Control flows sequentially.
  - The execution of a procedure starts at the beginning of the procedure body and ends at the point following where the procedure was called.
- Each execution of a procedure is referred to as an *activation* of the procedure. The *lifetime* of an activation is the sequence of steps between the first and last steps in the execution of the procedure body (including any other procedures called internally).
- In languages like Pascal, each time control enters a procedure Q from another P, control will eventually return to P in the absence of an error.
- So, if P and Q are procedure activations, their lifetimes are either nested or non-overlapping. That is if Q enters before P is left, then Q must terminate before P does.
- A procedure is recursive if a new activation can begin before an earlier activation of the same procedure finished (note: recursion may be indirect (P calls Q which calls P).

Activation Trees (2)

- We use a tree structure called an activation tree to depict this control flow. In this tree:
  - Each node represents an activation of a procedure.
  - The root node represents the activation of the main program procedure.
  - The node for A is the parent of another node B **iff** control flows from activation A to B.
  - The node for A is to the left of the node for B **iff** the lifetime of A occurs before that of B.

Example (1)

```
program sort
  var a: array[0..10] of integer;

  procedure readarray;
  var i:integer;
  begin
    for i := 1 to 9 do read(a[i]);
  end;

  function partition(y,z:integer):integer;
  var ...
  begin
    ...
  end;
```

Example (2)

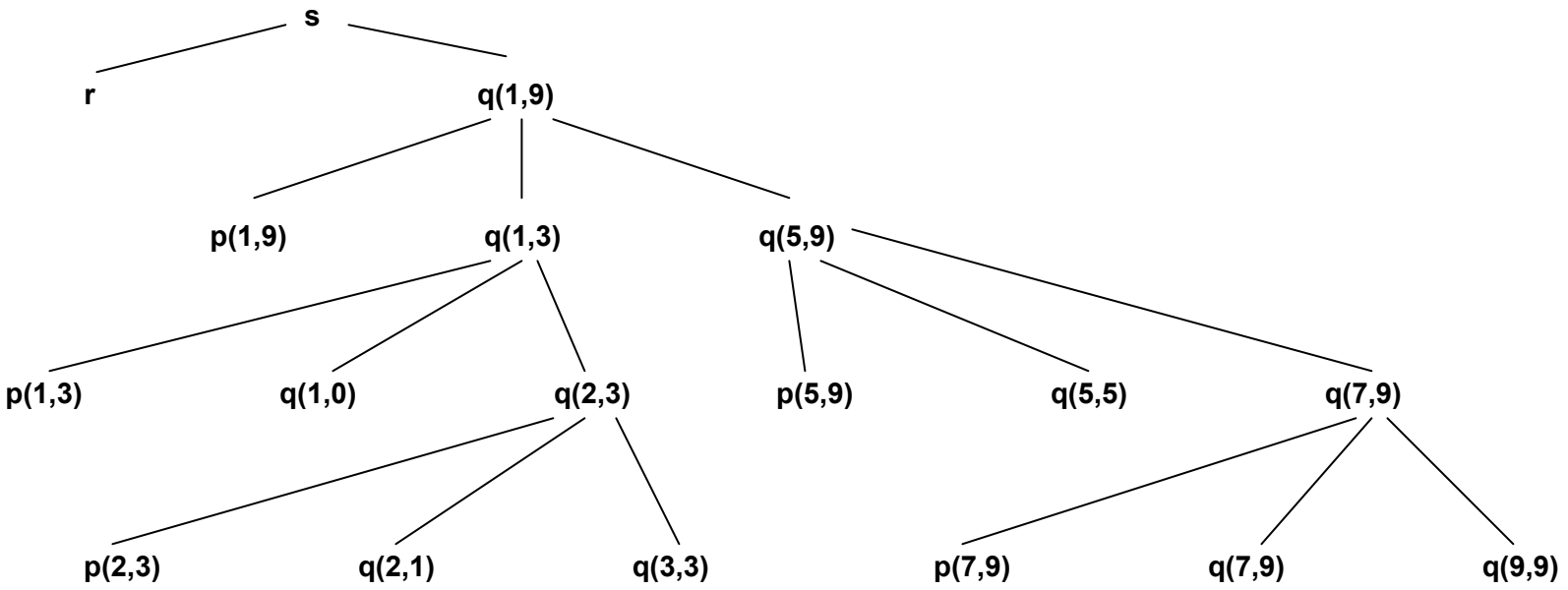
```
procedure quicksort(m,n:integer);
var i:integer;
begin
  if (n > m) then begin
    i := partition(m,n);
    quicksort(m,i-1);
    quicksort(i+1,n);
  end;
end;

begin
  a[0] := -9999; a[10] := 9999;
  readarray;
  quicksort(1,9);
end.
```

Example (3)

- Activation Trace:

```
Execution Begins
Enter readarray
Leave readarray
Enter quicksort(1,9)
Enter partition(1,9)
Leave partition(1,9)
Enter quicksort(1,3)
...
Leave quicksort(1,3)
Enter quicksort(5,9)
...
Leave quicksort(5,9)
Leave quicksort(1,9)
Execution Finishes
```

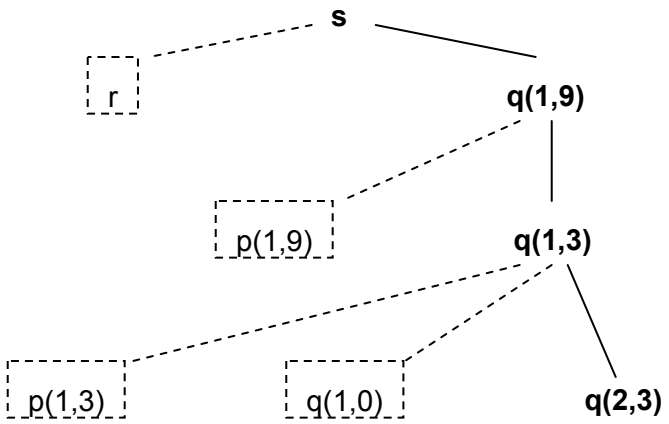
Example (4)


s
r
q(1,9)
p(1,9)
q(1,3)
q(5,9)
p(1,3)
q(1,0)
q(2,3)
p(5,9)
q(5,5)
q(7,9)
p(2,3)
q(2,1)
q(3,3)
p(7,9)
q(7,9)
q(9,9)

Control Stacks (1)

- The flow of control of a program corresponds to a depth first traversal of the activation tree.
  - (starts at the root, visits nodes before children and visits children in a left-to-right order)
- The trace we have seen before can be reconstructed by traversing the previous tree as illustrated above.
- We can use a stack called the **control stack** to keep track of live procedure activations. The idea is to push a node onto the stack when activation begins and popping it off when activation ends.
- When a node n is on top of the control stack, the stack contains the nodes along the path from n to the root (start).

Control Stacks (2)



The state of the stack when q(2,3) is on top.

Activation Records

- Information (memory space) needed for the execution of a single procedure is managed by a block of storage called an **activation record**.
- Not all languages or compilers use the same structure for this record.
- Common fields in this record are:
  - Temporaries: temporary values such as those intermediate values when evaluating an expression.
  - Local data: local values to the procedures.
  - Saved machine status: the state just before the procedure was called.
  - Access link: link to non-local data.
  - Control link: link to the activation record of the calling procedure.
  - Actual parameters: values of the actual parameters passed the procedure.
  - Returned value: the returned value if the procedure is a ‘function’.
- ‘Out of Stack Space’ issue in infinitely recurring calls.
- The sizes of most fields are usually determined at compile time with exceptions if there is a local array whose size depends on an actual argument or the procedure can take a variable number of parameters.

Intermediate Code

- It is common practice for the front end of a compiler to produce an intermediate form of code before passing that on to the backend to generate the target code itself.
- This is desirable since:
  - Retargeting is facilitated (a compiler for the same language but different machine).
  - A machine-independent code optimiser may be developed (optimisation applied to the intermediate code).
- We will assume that at this point the language has been parsed and statically checked.

Intermediate Languages (1)

- Syntax trees and postfix are two types of intermediate representations.
- In this section we will discuss a new one called the **three address code**.
- The three address code (3AC – *my abbreviation!*) is a sequence of statements of the general form:

  `x := y` *`op`* `z`

- Where x, y and z are names, constants or compiler-generated temporaries.
- Op, stands for an operator such as integer or floating-point arithmetic operators or a logical operator on boolean data.

Intermediate Languages (2)

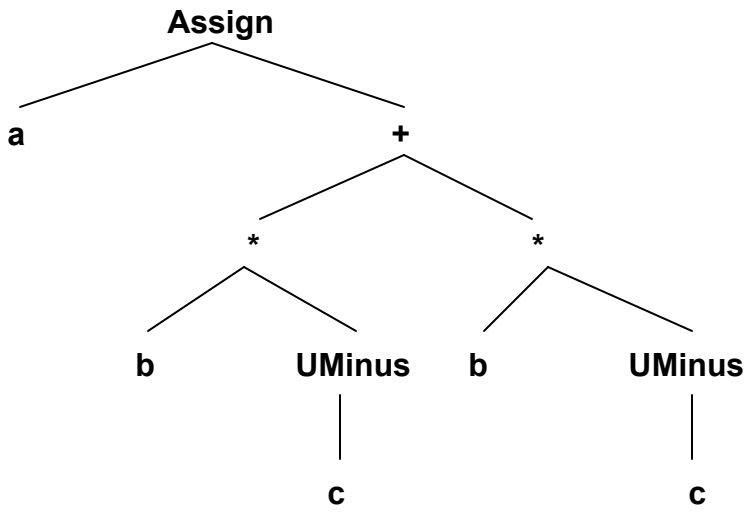
- Note that no ‘built-up’ expressions are allowed since there is only one operator in the RHS. So, something like `p + q * r`, would look like:

  t_1 := q * r

  t_2 := p + t_1
- Where, t_1 and t_2 are compiler-generated temporaries.
- The use of names for intermediate values allows 3AC to be easily rearrange unlike postfix notation.
- 3AC is a linear representation of the syntax tree (like postfix).
- The reason for the term ‘Three Address Code’ is that each statement **usually** contains 3 addresses, 2 for the operands and 1 for the result.

Intermediate Languages (3)

Syntax Tree



3AC

```
t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

a := b * -c + b * -c

Types of 3AC (1)

Type	Form	Notes
Assignment	x := y op z	Op is a binary arithmetic or logical operator.
Assignment	x := op y	Op is a unary operator (-, NOT,…)
Copy	x := y	Copy y into x.
Unconditional Jump	goto L	Where L is a label to the next statement to run.
Conditional Jump	if x relop y goto L	Apply a relational operator (>, <, <=,…) to x and y and jump to L if true otherwise continues with the next code.
Parameters, Calls and Returns	param x call p, *n* return y	y is an optional return value. Typically used as a sequence: param x1 … param x*n* call p,*n*

Types of 3AC (2)

Type	Form	Notes
Indexed Assignments	x := y[i] x[i] = y	The first sets x to the value in the location i memory units beyond y. The second sets the value at the location i memory units beyond x to y.
Address/Pointer Assignments	x := &y x := *y *x = y	The first sets the value of x to the memory location of y. In the second, presumably y is a pointer. In the third, presumably x is a pointer.

Note:
The operator set in the design of the 3AC must be rich enough to describe the operations in the source language. A small set is easier to implement on the target machine, but the resulting code would be very long, making the life of the optimiser harder if it is to produce good code.

Code Generation (1)

- The final phase in compiler is the code generator which takes an intermediate representation of a source program and generates equivalent target code.
- In between the intermediate code stage and code generation stage there could be a code optimisation stage. Code optimisation may be implemented on the final target code too.
- The requirements generally imposed on a code generator are that the target code should be correct of high quality and effectively use resources on the target computer. Also the code generator itself must be efficient.
- Mathematically, the problem of generating optimal code is undecidable. In practice, heuristics that generate good code (not necessarily optimal) are typically used.
- The choice of such heuristics is important. Carefully designed code generators may produce code that is several times faster than that produced by a bad one.

Code Generation (2)

- In order to design an efficient code generator the designer must have intimate knowledge of the target hardware and operating system.
- Issues such as memory management, instruction selection, register allocation and evaluation order are inherent to almost all code generation problems.
- Due to highly specialised, platform-dependent issues, in this section will examine generic design issues only.

Input to the Code Generator

- The input is typically,
  - The intermediate code produced by the front end.
  - The symbol table that is used to determine the runtime addresses of the data objects denoted by the names in the intermediate representation.
- We assume that,
  - The source code has been properly scanned and parsed correctly.
  - All relevant information is available to the code generator.
  - Type checking has occurred.
  - In general the input is error-free.

Target Programs

- The output of a code generator is the target language. This may take on different forms such as,
  - Absolute machine code,
  - Relocatable machine language,
  - Or assembly language.
- Producing absolute machine code has the advantage that it can be placed in a fixed memory location and execute immediately.
- Producing relocatable (**object**) code has the advantage that separate sub-programs may be compiled separately and then linked and loaded to execute. Whilst the code has the overhead of linking and loading we gain a lot of flexibility (think DLLs – though not exactly).
- Producing assembly language makes the code generation task simpler but involves the extra step of assembling the output.

Memory Management

- Mapping names in the source program to addresses of data objects is done cooperatively by the front-end and back-end of the compiler.
- A name in a 3AC statement refers to a symbol table entry for the name.
- The type of a declaration determines the amount of storage allocated in memory (e.g. a long integer would take up 4 bytes).

Instruction Selection (1)

- The nature of the instruction set of the target machines determines the instruction selection when generating code.
- Also, if the target machine does not support each data type natively, special arrangements have to be made.
- Instruction speeds are an important factors when generating code. If the quality of the target code is not an issue, then each 3AC statement could be associated with a 'template'. For example, every 3AC statement of the form `x := y + z` could be translated into:

```
MOV y, R0
ADD z, R0
MOV R0, x
```

Instruction Selection (2)

- Unfortunately, this technique can (and most likely will) produce inefficient code. For example:

```
a := b + c
d := a + e
```

- Will produce:

```
MOV b,R0
ADD c,R0
MOV R0,a
MOV a,R0
ADD e,R0
MOV R0,d
```

- There the third and fourth statements are redundant if a is not subsequently used.

Instruction Selection (3)

- The quality of code is usually determined by its execution speed and its size.
- A target machine with a rich instruction set may provide several ways to perform any given operation. In this case a ‘narrow-minded’ code generator may produce correct, though unacceptably inefficient code.
- For example, say, the machine supports an increment (INC) operation. Then the 3AC instruction a := a + 1 may be implemented more efficiently by the single instruction rather than using the ‘template’ we have seen before.

Register Allocation

- Instructions involving register operands are usually shorter and much faster than those involving memory operands. For this reason, efficient utilization of registers is important in generating fast code.
- The use of registers is often subdivided into two problems:
  - During **register allocation**, we select the set of variables that will reside in registers at a point in the program.
  - During subsequent **register assignment**, we pick the specific register that the variable will ‘live’ in.
- Finding the optimal assignment of registers is mathematically NP complete and further restrictions may be enforced by the hardware and/or operating system

Code Optimisation

- An optimiser looks at a representation of the source program and tries to produce **shorter** or **faster** code (or both).
- There are essentially two ways in which optimisation can take place:
  - Reorganise the structure of the source algorithms to make them more efficient. This generally operates on the parse tree. This technique is machine independent.
  - Modification of the code produced by a simple translator to make it efficient. This phase operates on the object code.

Common Optimisation Tasks (1)

- Common Sub Expressions
  - An occurrence of an expression E is called a **common sub-expression** if E was previously computed and the values of the variables in E have not changed. In such cases we can avoid recomputing an expression of we can use the previously computed value.
- Copy Propagation
  - Reorganises assignment statements so that:

```
x = y
z = x
```

  - becomes

```
x = y
z = y
```

  - (more on this later)

Common Optimisation Tasks (2)

- Dead Code Elimination
  - A variable is ‘live’ at a point in a program if its value can be used subsequently, otherwise it is ‘dead’ at that point. Statements may compute values that may never be used in a program. While a programmer is unlikely to introduce dead code intentionally, it may appear as a result of previous transformations. Consider the statement:

    ```
    if (debug) then Print ...
    ```

  - By data flow analysis it may be deduced that no matter what path the program takes, when the statement is reached, the value of `debug` would always be false, so the test and printing may be removed from the object code.

Common Optimisation Tasks (3)

- Dead Code Elimination Continued
  - One advantage of copy propagation is that it often turns an assignment statement into dead code. For example copy propagation followed by dead code elimination would convert:

```
x = t3
a[t2] = t5
a[t4] = x
goto b2
```

  - By elimination of copy propagation:

```
x = t3
a[t2] = t5
a[t4] = t3
goto b5
```

  - By Dead code elimination:

```
a[t2] = t5
a[t4] = t3
goto b5
```

Common Optimisation Tasks (4)

- Loop Optimisation
  - Loops are an important place where optimisations may occur. The running time of a loop may be improved if we decrease the number of instructions occurring inside. A common loop optimisation is called code motion.
  - Code motion attempts to move code out of the loop (though the expression must yield the same result). This transformation takes an expression that has the same evaluation independent of the number of times the loop executes (called a loop-invariant computation) and places it before the loop. For example the computation of `limit - 1`, is loop invariant in:

    ```
    while (i < (limit - 1)) ...
    ```

  - So we can have:

    ```
    t = limit - 1
    while (i < t) ...
    ```