CS432F/CSSL 728: Compiler Design

July 2004

S. Arun-Kumar

sak@cse.iitd.ernet.in

Department of Computer Science and Engineering
I. I. T. Delhi, Hauz Khas, New Delhi 110 016.

July 30, 2004
Contents

• first The Bigpicture last
• first Lexical Analysis last
• first Syntax Analysis last
  – first Context-free Grammars last
  – first Ambiguity last
  – first Shift-Reduce Parsing last
  – first Parse Trees last
• first Semantic Analysis last
• first Synthesized Attributes last
• first Inherited Attributes last

Contd . . .
Contents

... Contd

- \textit{first} Abstract Syntax Trees \textit{last}
- \textit{first} Symbol Tables \textit{last}
- \textit{first} Intermediate Representation \textit{last}
  - IR: Properties
  - Typical Instruction Set
  - IR: Generation
- \textit{first} Runtime Structure \textit{last}
The Big Picture: 1

- stream of characters
- scanner
- stream of tokens
The Big Picture: 2

- stream of characters
- stream of tokens
- parse tree

SCANNER

PARSER
The Big Picture: 3

- **SCANNER**
  - stream of characters
  - stream of tokens

- **PARSER**
  - parse tree

- **SEMANTIC ANALYZER**
  - abstract syntax tree
The Big Picture: 4

- **SCANNER**
  - stream of characters

- **PARSER**
  - stream of tokens
  - parse tree

- **SEMANTIC ANALYZER**
  - abstract syntax tree

- **I.R. CODE GENERATOR**
  - intermediate representation
The Big Picture: 5

- **SCANNER**
  - stream of characters
- **PARSER**
  - stream of tokens
  - parse tree
- **SEMANTIC ANALYZER**
  - abstract syntax tree
- **I.R. CODE GENERATOR**
  - intermediate representation
- **OPTIMIZER**
  - optimized intermediate representation
The Big Picture: 6

- Scanner
  - Stream of characters

- Parser
  - Stream of tokens
  - Parse tree

- Semantic Analyzer
  - Abstract syntax tree

- I.R. Code Generator
  - Intermediate representation

- Optimizer
  - Optimized intermediate representation

- Code Generator
  - Target code
The Big Picture: 7

- Scanner
- Parser
- Semantic Analyzer
- I.R. Code Generator
- Optimizer
- Code Generator
- Symbol Table Manager

- Stream of characters
- Stream of tokens
- Parse tree
- Abstract syntax tree
- Intermediate representation
- Optimized intermediate representation
- Target code
The Big Picture: 7

Scanner  Parser  Semantic Analysis  Symbol Table
IR  Optimization  Contents
Scanning: 1

- Takes a stream of characters and identifies tokens from the lexemes.

- Eliminates comments and redundant whitespace.

- Keeps track of line numbers and column numbers and passes them as parameters to the other phases to enable error-reporting to the user.
Scanning: 2

- **Whitespace**: A sequence of space, tab, newline, carriage-return, form-feed characters etc.

- **Lexeme**: A sequence of non-whitespace characters delimited by whitespace or special characters (e.g. operators like +, -, *).

- Examples of lexemes.
  - reserved words, keywords, identifiers etc.
  - Each comment is usually a single lexeme
  - preprocessor directives
Scanning: 3

- **Token**: A sequence of characters to be treated as a single unit.
- **Examples of tokens.**
  - Reserved words (e.g. `begin`, `end`, `struct`, `if` etc.)
  - Keywords (`integer`, `true` etc.)
  - Operators (`+`, `&&`, `++` etc)
  - Identifiers (variable names, procedure names, parameter names)
  - Literal constants (numeric, string, character constants etc.)
  - Punctuation marks (`,`, `;`, `etc.`)
Scanning: 4

- Identification of tokens is usually done by a Deterministic Finite-state automaton (DFA).
- The set of tokens of a language is represented by a large regular expression.
- This regular expression is fed to a lexical-analyser generator such as Lex, Flex or ML-Lex.
- A giant DFA is created by the Lexical analyser generator.
The Big Picture
Syntax Analysis

Consider the following two languages over an alphabet $A = \{a, b\}$.

\[
R = \{a^n b^n | n < 100\}
\]

\[
P = \{a^n b^n | n > 0\}
\]

- $R$ may be finitely represented by a regular expression (even though the actual expression is very long).

- However, $P$ cannot actually be represented by a regular expression

- A regular expression is not powerful enough to represent languages which require parenthesis matching to arbitrary depths.

- All high level programming languages require an underlying language of expressions which require parentheses to be nested and matched to arbitrary depth.
CF-Grammars: Definition

A context-free grammar (CFG) \( G = \langle N, T, P, S \rangle \) consists of

- a set \( N \) of nonterminal symbols,
- a set \( T \) of terminal symbols or the alphabet,
- a set \( P \) of productions or rewrite rules,
- each production is of the form \( X \rightarrow \alpha \), where
  - \( X \in N \) is a nonterminal and
  - \( \alpha \in (N \cup T)^* \) is a string of terminals and nonterminals
- a start symbol \( S \in N \).
CFG: Example

\[ G = \langle \{ S \}, \{ a, b \}, P, S \rangle, \text{ where } S \rightarrow ab \text{ and } S \rightarrow aSb \]

are the only productions in \( P \).

Derivations look like this:

\[
\begin{align*}
S & \Rightarrow ab \\
S & \Rightarrow aSb \Rightarrow aabb \\
S & \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb
\end{align*}
\]

\( \mathcal{L}(G) \), the language generated by \( G \) is \( \{ a^n b^n | n > 0 \} \).

Actually can be proved by induction on the length and structure of derivations.
CFG: Empty word

\[ G = \langle \{ S \}, \{ a, b \}, P, S \rangle, \] where \( S \rightarrow SS \mid aSb \mid \varepsilon \)
generates all sequences of matching nested parentheses, including the empty word \( \varepsilon \).

A leftmost derivation might look like this:

\[
S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \ldots
\]

A rightmost derivation might look like this:

\[
S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow SaSb \Rightarrow Sab \Rightarrow aSbab \ldots
\]

Other derivations might look like \textit{God alone knows what}!

\[
S \Rightarrow SS \Rightarrow SSS \Rightarrow SS \Rightarrow \ldots
\]

Could be quite confusing!
CFG: Derivation trees

Derivation sequences

- put an artificial order in which productions are fired.
- instead look at trees of derivations in which we may think of productions as being fired in parallel.
- There is then no highlighting in red to determine which copy of a nonterminal was used to get the next member of the sequence.
- Of course, generation of the empty word $\varepsilon$ must be shown explicitly in the tree.
CFG: Derivation trees

Derivation tree of

$abaabb$
Another Derivation tree of $abaabb$
Yet another Derivation tree of \textit{abaabb}
Consider the sentence $y + 4 \ast z$. 

\begin{align*}
E & \rightarrow I \mid C \mid E+E \mid E\ast E \\
I & \rightarrow y \mid z \\
C & \rightarrow 4
\end{align*}
Ambiguity: 2

\[ E \rightarrow I \mid C \mid E+E \mid E*E \]
\[ I \rightarrow y \mid z \]
\[ C \rightarrow 4 \]

Consider the sentence \( y + 4 * z \).
Ambiguity: 3

\[ E \rightarrow I \mid C \mid E+E \mid E*E \]
\[ I \rightarrow y \mid z \]
\[ C \rightarrow 4 \]

Consider the sentence \( y + 4 \ast z \).
Consider the sentence $y + 4 * z$. 
Consider the sentence $y + 4 * z$. 

Diagram: 

- $E \rightarrow I \mid C \mid E+E \mid E*E$
- $I \rightarrow y \mid z$
- $C \rightarrow 4$

Ambiguity: 5
r1. E → E − T
r2. E → T
r3. T → T / D
r4. T → D
r5. D → a | b | ( E )

a − a / b
Principle:
Reduce whenever possible.
Shift only when reduce is impossible

r1. $E \rightarrow E - T$

r2. $E \rightarrow T$

r3. $T \rightarrow T / D$

r4. $T \rightarrow D$

r5. $D \rightarrow a | b | ( E )$

- $a / b$
r1. $E \rightarrow E - T$

r2. $E \rightarrow T$

r3. $T \rightarrow T / D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid (E)$

Reduce by r5
Parsing: 3

r1. $E \rightarrow E \cdot T$

r2. $E \rightarrow T$

r3. $T \rightarrow T / D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid (E)$

Reduce by r4
r1. \( E \rightarrow E - T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T / D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mid b \mid (E) \)

Reduce by r2
r1. \( E \rightarrow E \cdot T \)

r2 \( E \rightarrow T \)

r3 \( T \rightarrow T \, / \, D \)

r4 \( T \rightarrow D \)

r5 \( D \rightarrow a \mid b \mid ( \, E \, ) \)

Shift
r1. $E \rightarrow E - T$

r2. $E \rightarrow T$

r3. $T \rightarrow T \, / \, D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid (E)$

The diagram shows a parsing tree with rules for expanding the non-terminal symbols $E$, $T$, and $D$. The tree is colored with symbols $a$, $b$, and $E$, and it indicates a shift operation at the bottom.
r1. E → E − T
r2. E → T
r3. T → T / D
r4. T → D
r5. D → a | b | ( E )

Reduce by r5
r1. \( E \rightarrow E \ T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T \ D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \ | \ b \ | \ (E) \)

Reduce by r4
Parsing: 8a

r1. $E \rightarrow E \cdot T$

r2. $E \rightarrow T$

r3. $T \rightarrow T / D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid (E)$

Reduce by $r4$
Parsing: 9a

r1. $E \rightarrow E \rightarrow T$

r2. $E \rightarrow T$

r3. $T \rightarrow T \rightarrow D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid (E)$

Reduce by r1
r1. \[ E \rightarrow E - T \]

r2. \[ E \rightarrow T \]

r3. \[ T \rightarrow T / D \]

r4. \[ T \rightarrow D \]

r5. \[ D \rightarrow a \mid b \mid (E) \]

Shift
Parsing: 11a

r1. \( E \rightarrow E \cdot T \)

r2 \( E \rightarrow T \)

r3 \( T \rightarrow T \cdot D \)

r4 \( T \rightarrow D \)

r5 \( D \rightarrow a \mid b \mid ( E ) \)

\( \text{Shift} \)
r1. \( E \rightarrow E - T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T / D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mid b \mid (E) \)

Reduce by r5
Parsing: 13a

r1. \( E \rightarrow E \cdot T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T \cdot D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mid b \mid (E) \)

Reduce by r4
Parsing: 14a

r1. E → E T
r2. E → T
r3. T → T / D
r4. T → D
r5. D → a | b | ( E )

Get back!

Reduce by r2

Stuck!
Parsing: 14b

r1. \[ E \rightarrow E - T \]
r2. \[ E \rightarrow T \]
r3. \[ T \rightarrow T \frac{r_2}{/} D \]
r4. \[ T \rightarrow D \]
r5. \[ D \rightarrow a \mid b \mid (E) \]

Get back!  Reduce by r2
r1. \( E \rightarrow E - T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T / D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mid b \mid (E) \)

Get back!

Reduce by r4
r1. \( E \rightarrow E \rightarrow T \)

r2. \( E \rightarrow T \)

r3. \( T \rightarrow T \rightarrow D \)

r4. \( T \rightarrow D \)

r5. \( D \rightarrow a \mid b \mid (E) \)

Get back!

Reduce by r5
Parsing: 11b

r1. \( E \rightarrow E \mathbf{-} T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T / D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a | b | (E) \)

Get back!  Shift
Parsing: 10b

r1. \( E \rightarrow E \cdot T \)

r2 \( E \rightarrow T \)

r3 \( T \rightarrow T \cdot D \)

r4 \( T \rightarrow D \)

r5 \( D \rightarrow a \mid b \mid (E) \)

Get back!  Shift
Get back to where you once belonged!

Reduce by r1
Principle: Modified
Reduce whenever possible, but
but depending upon 
lookahead
Shift instead of reduce here!
Shift–reduce conflict
Reduce by r4
r1. E → E - T
r2. E → T
r3. T → T / D
r4. T → D
r5. D → a | b | ( E )

Reduce by r4
r1. \( E \rightarrow E\ -\ T \)

r2 \( E \rightarrow T \)

r3 \( T \rightarrow T\ \slash\ D \)

r4 \( T \rightarrow D \)

r5 \( D \rightarrow a \mid b \mid (\ E \ ) \)

Shift
r1. \( E \rightarrow E \rightarrow T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T \rightarrow D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mid b \mid (E) \)

Shift
r1. \( E \rightarrow E \rightarrow T \)

r2. \( E \rightarrow T \)

r3. \( T \rightarrow T \rightarrow D \)

r4. \( T \rightarrow D \)

r5. \( D \rightarrow a \mid b \mid (E) \)

Reduce by \( r5 \)
r1.  $E \rightarrow E \rightarrow T$

r2.  $E \rightarrow T$

r3.  $T \rightarrow T \rightarrow D$

r4.  $T \rightarrow D$

r5.  $D \rightarrow a \mid b \mid (E)$

Reduce by $r3$
r1. $E \rightarrow E \ + \ T$

r2. $E \rightarrow T$

r3. $T \rightarrow T \ / \ D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \ | \ b \ | \ (E)$

Reduce by r1
Parse Trees: 0

r1. E → E * T
r2. E → T
r3. T → T / D
r4. T → D
r5. D → a | b | (E)

```
  a
  -
  a / b
```
Parse Trees: 1

1. \( E \rightarrow E - T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \lor D \)
4. \( T \rightarrow D \)
5. \( D \rightarrow a \lor b \lor (E) \)

Diagram:

- **D**: Non-terminal symbol
- **a**, **b**: Terminal symbols
- **\ lor **: Operator symbol
- **\rightarrow**: Production rule
Parse Trees: 2

r1. $E \rightarrow E - T$

r2. $E \rightarrow T$

r3. $T \rightarrow T \mid D$

r4. $T \rightarrow D$

r5. $D \rightarrow a \mid b \mid \{ E \}$
Parse Trees: 3

r1. \( E \rightarrow E - T \)

r2. \( E \rightarrow T \)

r3. \( T \rightarrow T \lor D \)

r4. \( T \rightarrow D \)

r5. \( D \rightarrow a \mid b \mid (E) \)
Parse Trees: 3a

r1. \[ E \rightarrow E - T \]
r2. \[ E \rightarrow T \]
r3. \[ T \rightarrow T / D \]
r4. \[ T \rightarrow D \]
r5. \[ D \rightarrow a \mid b \mid (E) \]
Parse Trees: 3b

r1. \( E \rightarrow E - T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow (D \lor T) \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \lor b \lor (E) \)
Parse Trees: 4

r1. E \rightarrow E \cdot T
r2. E \rightarrow T
r3. T \rightarrow T \cdot D
r4. T \rightarrow D
r5. D \rightarrow a \mid b \mid (E)

E
  /\n /\n /\n a

T
  /\n  /\n  /\n a

D
  /\n  /\n  /\n a

D
  /\n  /\n  /\n /

b
Parse Trees: 5

r1. E → E − T
r2. E → T
r3. T → T (/) D
r4. T → D
r5. D → a | b | (E)

Diagram:

```
  E
   \-- T
       \-- D
           a
           └── /
               b
```

```
  T
   \-- D
       a
       └── /
           b
```

```
  T
      \-- (/)
          a
```

```
  T
      \-- (/)
          b
```

```
  T
      \-- (/)
          (E)
```

Parse Trees: 5a

- $E \rightarrow E + T$
- $E \rightarrow T$
- $T \rightarrow T D$
- $T \rightarrow D$
- $D \rightarrow a | b | (E)$
Parse Trees: 5b

r1. \( E \rightarrow E - T \)
r2. \( E \rightarrow T \)
r3. \( T \rightarrow T / D \)
r4. \( T \rightarrow D \)
r5. \( D \rightarrow a \mid b \mid \{ E \} \)
Parse Trees: 6

1. \( E \rightarrow E \; T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T \; D \)
4. \( T \rightarrow D \)
5. \( D \rightarrow a \; | \; b \; | \; ( E ) \)

Diagram:

- \( E \)
- \( T \)
- \( D \)
- \( a \)
- \( b \)
Parse Trees: 7

r1. E → E T
r2. E → T
r3. T → T D
r4. T → D
r5. D → a | b | (E)

```
E
  T
  D
  a
```
```
T
  T
  D
  a
```
```
T
  T
  D
  /
  a
  b
```
```
D
  D
  (E)
```
Parse Trees: 8

\[
\begin{align*}
    r1. & \quad E \rightarrow E - T \\
    r2. & \quad E \rightarrow T \\
    r3. & \quad T \rightarrow T \lor D \\
    r4. & \quad T \rightarrow D \\
    r5. & \quad D \rightarrow a \mid b \mid \{E\}
\end{align*}
\]
Parsing: Summary: 1

- All high-level languages are designed so that they may be parsed in this fashion with only a single token look-ahead.
- Parsers for a language can be automatically constructed by parser-generators such as Yacc, Bison, ML-Yacc.
- Shift-reduce conflicts if any, are automatically detected and reported by the parser-generator.
- Shift-reduce conflicts may be avoided by suitably redesigning the context-free grammar.
Parsing: Summary: 2

• Very often shift-reduce conflicts may occur because of the prefix problem. In such cases many parser-generators resolve the conflict in favour of shifting.

• There is also a possibility of reduce-reduce conflicts. This usually happens when there is more than one non-terminal symbol to which the contents of the stack may reduce.

• A minor reworking of the grammar to avoid redundant non-terminal symbols will get rid of reduce-reduce conflicts.

The Big Picture
Semantic Analysis: 1

- Every Programming language can be used to program any computable function, assuming of course, it has
  - unbounded memory, and
  - unbounded time

- The parser of a programming language provides the framework within which the target code is to be generated.

- The parser also provides a structuring mechanism that divides the task of code generation into bits and pieces determined by the individual nonterminals and production rules.

- However, context-free grammars are not powerful enough to represent all computable functions. Example, the language \( \{a^n b^n c^n | n > 0\} \).
Semantic Analysis: 2

- There are context-sensitive aspects of a program that cannot be represented/enforced by a context-free grammar definition. Examples include
  - correspondence between formal and actual parameters
  - type consistency between declaration and use.
  - scope and visibility issues with respect to identifiers in a program.
Synthesized Attributes:

0
Synthesized Attributes: 1

Synthesized Attributes

1 2 3 4
Synthesized Attributes:

2

Synthesized Attributes

4 3 2 1
Synthesized Attributes:

3
Synthesized Attributes:

4
Synthesized Attributes: 5
Synthesized Attributes: 6
Synthesized Attributes: 7

The diagram illustrates a tree structure with labeled nodes. Each node is marked with values of E (End), T (True), F (False), and numbers indicating a synthesized attribute value. The tree branches out with the following nodes:

- E
- T
- F
- 4
- 3
- 2
- 1
Synthesized Attributes:

8
Synthesized Attributes:

\[ ET \quad F \]
\[ T \]
\[ T \quad F \]
\[ / \]
\[ / \]
\[ ( ) \]
\[ ( ) \]
\[ n \]
\[ n \]
\[ n \]
\[ n \]

Synthesized Attributes
Synthesized Attributes: 10
Synthesized Attributes:

11
Synthesized Attributes: 12
Synthesized Attributes: 13
Synthesized Attributes:

14

Synthesized Attributes
An Attribute Grammar

\[
E_0 \rightarrow E_1 - T \quad \Rightarrow \quad E_0.val := sub(E_1.val, T.val)
\]

\[
E \rightarrow T \quad \Rightarrow \quad E.val := T.val
\]

\[
T_0 \rightarrow T_1 / F \quad \Rightarrow \quad T_0.val := div(T_1.val, F.val)
\]

\[
T \rightarrow F \quad \Rightarrow \quad T.val := F.val
\]

\[
F \rightarrow (E) \quad \Rightarrow \quad F.val := E.val
\]

\[
F \rightarrow n \quad \Rightarrow \quad F.val := n.val
\]
$C$-style declarations generating $\text{int } x, y, z$.

\[
D \rightarrow TL \\
L \rightarrow L, I | I \\
T \rightarrow \text{int } | \text{float} \\
I \rightarrow x | y | z
\]
C-style declarations generating \( \text{int} \ x, \ y, \ z. \)

\[
D \rightarrow T \ L \\
L \rightarrow L, I \mid I
\]

\[
T \rightarrow \text{int} \mid \text{float} \\
I \rightarrow x \mid y \mid z
\]
Inherited Attributes: 2

C-style declarations generating \texttt{int x, y, z}.

\[
\begin{align*}
D & \rightarrow T \ L \\
L & \rightarrow L, I \mid I \\
T & \rightarrow \text{int} \mid \text{float} \\
I & \rightarrow x \mid y \mid z
\end{align*}
\]
\[ D \rightarrow T \ L \ \\
L \rightarrow L, I \ | \ I \ \\
T \rightarrow \text{int} \ | \ \text{float} \ \\
I \rightarrow x \ | \ y \ | \ z \]
Inherited Attributes: 4

C-style declarations generating \( \text{int} \ x, y, z. \)

\[
D \rightarrow TL \\
L \rightarrow L, I | I
\]

\[
T \rightarrow \text{int} | \text{float} \\
I \rightarrow x | y | z
\]
Inherited Attributes: 5
Inherited Attributes: 6

C-style declarations generating \texttt{int \ x, \ y, \ z}.

\[ D \rightarrow TL \]
\[ L \rightarrow L,l \mid I \]

\[ T \rightarrow \text{int} \mid \text{float} \]
\[ I \rightarrow x \mid y \mid z \]
Inherited Attributes: 7

C-style declarations generating \texttt{int x, y, z}.

\[ D \rightarrow T \ L \]
\[ L \rightarrow L, I \mid I \]
\[ T \rightarrow \text{int} \mid \text{float} \]
\[ I \rightarrow x \mid y \mid z \]
An Attribute Grammar

\[ D \rightarrow TL \quad \triangleright \quad L.in := T.type \]

\[ T \rightarrow \text{int} \quad \triangleright \quad T.type := \text{int}.int \]

\[ T \rightarrow \text{float} \quad \triangleright \quad T.type := \text{float}.float \]

\[ L_0 \rightarrow L_1,I \quad \triangleright \quad L_1 := L_0.in \]

\[ L \rightarrow I \quad \triangleright \quad I.in := L.in \]

\[ I \rightarrow \text{id} \quad \triangleright \quad \text{id}.type := I.in \]
Abstract Syntax: 0

\[
E \rightarrow E - T \mid T \\
T \rightarrow T / F \mid F \\
F \rightarrow n \mid (E)
\]

Suppose we want to evaluate an expression \((4 - 1)/2\). What we actually want is a tree that looks like this:
Evaluation: 0
Evaluation: 1

Diagram:

- Root labeled 1
- Branches leading to numbers 4 and 2
- Number 4 is inside a circle
- Number 2 is outside the circle
Evaluation: 2

/  
3  2
Evaluation: 3

Diagram:

```
/          \
3          2
```

Evaluation: 3/3
But what we *actually get* during parsing is a tree that looks like . . .
Abstract Syntax: 1

... THIS!
Abstract Syntax: 2

We use attribute grammar rules to construct the abstract syntax tree (AST)!. But in order to do that we first require two procedures for tree construction.

**makeLeaf**(`literal`) : Creates a node with label `literal` and returns a pointer to it.

**makeBinaryNode**(`opr`, `opd1`, `opd2`) : Creates a node with label `opr` (with fields which point to `opd1` and `opd2`) and returns a pointer to the newly created node.

Now we may associate a synthesized attribute called `ptr` with each terminal and nonterminal symbol which points to the root of the subtree created for it.
Abstract Syntax: 3

\[E_0 \rightarrow E_1 - T \Downarrow E_0.\text{ptr} := \text{makeBinaryNode}(-, E_1.\text{ptr}, T.\text{ptr})\]

\[E \rightarrow T \Downarrow E.\text{ptr} := T.\text{ptr}\]

\[T_0 \rightarrow T_1 / F \Downarrow T_0.\text{ptr} := \text{makeBinaryNode}(/, T_1.\text{ptr}, F.\text{ptr})\]

\[T \rightarrow F \Downarrow T.\text{ptr} := F.\text{ptr}\]

\[F \rightarrow (E) \Downarrow F.\text{ptr} := E.\text{ptr}\]

\[F \rightarrow n \Downarrow F.\text{ptr} := \text{makeLeaf}(n.\text{val})\]

The Big Picture
Symbol Table:1

- The store house of **context-sensitive** and **run-time** information about every identifier in the source program.
- All accesses relating to an identifier require to first find the **attributes** of the identifier from the symbol table.
- Usually organized as a **hash** table – provides fast access.
- Compiler-generated temporaries may also be stored in the symbol table.
Symbol Table:2

Attributes stored in a symbol table for each identifier:

- type
- size
- scope/visibility information
- base address
- addresses to location of auxiliary symbol tables (in case of records, procedures, classes)
- address of the location containing the string which actually names the identifier and its length in the string pool
Symbol Table:3

- A symbol table exists throughout the compilation and run-time.
- Major operations required of a symbol table:
  - insertion
  - search
  - deletions are purely logical (depending on scope and visibility) and not physical
- Keywords are often stored in the symbol table before the compilation process begins.
Symbol Table: 4

Accesses to the symbol table at every stage of the compilation process,

**Scanning:** Insertion of new identifiers.

**Parsing:** Access to the symbol table to ensure that an operand exists (declaration before use).

**Semantic analysis:**
- Determination of types of identifiers from declarations
- type checking to ensure that operands are used in type-valid contexts.
- Checking scope, visibility violations.
Symbol Table:5

**IR generation:** Memory allocation and relative\(^a\) address calculation.

**Optimization:** All memory accesses through symbol table

**Target code:** Translation of relative addresses to absolute addresses in terms of word length, word boundary etc.

---

\(^a\)i.e. relative to a base address that is known only at run-time
Intermediate Representation

Intermediate representations are important for reasons of portability.

• *(more or less) independent* of specific features of the high-level language.

  Example. Java byte-code for any high-level language.

• *(more or less) independent* of specific features of any particular target architecture (e.g. number of registers, memory size)
  – number of registers
  – memory size
  – word length
IR Properties: 1

1. It is fairly **low-level** containing instructions common to all target architectures and assembly languages.
   How low can you stoop? . . .

2. It contains some fairly **high-level** instructions that are common to most high-level programming languages.
   How high can you rise?

3. To ensure **portability**
   - an **unbounded** number of variables and memory locations
   - no commitment to **Representational Issues**

4. To ensure **type-safety**
   - memory locations are also typed according to the data they may contain,
   - no commitment is made regarding word boundaries, and the structure of individual data items.
IR: Representation?

- No commitment to word boundaries or byte boundaries
- No commitment to representation of
  - int vs. float,
  - float vs. double,
  - packed vs. unpacked,
  - strings – where and how?.

Back to IR Properties:1
IR: How low can you stoop?

- most arithmetic and logical operations, load and store instructions etc.
- so as to be interpreted easily,
- the interpreter is fairly small,
- execution speeds are high,
- to have fixed length instructions (where each operand position has a specific meaning).

Back to IR Properties:1
IR: How high can you rise?

- typed variables,
- temporary variables instead of registers,
- array-indexing,
- random access to record fields,
- parameter-passing,
- pointers and pointer management
- no limits on memory addresses

Back to IR Properties:1
A typical instruction set: 1

Three address code: A suite of instructions. Each instruction has at most 3 operands.

- an opcode representing an operation with at most 2 operands
- two operands on which the binary operation is performed
- a target operand, which accumulates the result of the (binary) operation.

If an operation requires less than 3 operands then one or more of the operands is made null.
A typical instruction set: 2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing
A typical instruction set: 2

- Assignments (LOAD-STORE)
  - $x := y \text{ bop } z$, where bop is a binary operation
  - $x := uop y$, where uop is a unary operation
  - $x := y$, load, store, copy or register transfer

- Jumps (conditional and unconditional)

- Procedures and parameters

- Arrays and array-indexing

- Pointer Referencing and Dereferencing
A typical instruction set: 2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
  - `goto L` - Unconditional jump,
  - `x relop y goto L` - Conditional jump, where `relop` is a relational operator
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing
A typical instruction set: 2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
  - `call p n`, where `n` is the number of parameters
  - `return y`, return value from a procedures call
  - `param x`, parameter declaration
- Arrays and array-indexing
- Pointer Referencing and Dereferencing
A typical instruction set: 2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
  - \( x := a[i] \) – array indexing for \textit{r-value}
  - \( a[j] := y \) – array indexing for \textit{l-value}

Note: The two opcodes are different depending on whether \textit{l-value} or \textit{r-value} is desired. \( x \) and \( y \) are always simple variables
- Pointer Referencing and Dereferencing
A typical instruction set: 2

- Assignments (LOAD-STORE)
- Jumps (conditional and unconditional)
- Procedures and parameters
- Arrays and array-indexing
- Pointer Referencing and Dereferencing
  - \( x := ^y \) – referencing: set \( x \) to point to \( y \)
  - \( x := *y \) – dereferencing: copy contents of location pointed to by \( y \) into \( x \)
  - \( ^x := y \) – dereferencing: copy \textit{r-value} of \( y \) into the location pointed to by \( x \)

Picture
Pointers

\[
x := ^y
\]

\[
x := *y
\]

\[
x := y
\]
IR: Generation

- Can be generated by recursive traversal of the abstract syntax tree.
- Can be generated by syntax-directed translation as follows:
  For every non-terminal symbol $N$ in the grammar of the source language there exist two attributes:
  - $N$.place, which denotes the address of a temporary variable where the result of the execution of the generated code is stored.
  - $N$.code, which is the actual code segment generated.
- In addition a global counter for the instructions generated is maintained as part of the generation process.
- It is independent of the source language but can express target machine operations without committing to too much detail.
IR: Infrastructure

Given an abstract syntax tree $T$, with $T$ also denoting its root node.

$T\.place$ address of $\text{temporary}$ variable where result of execution of the $T$ is stored.

$new\text{temp}$ returns a $\text{fresh}$ variable name and also installs it in the symbol table along with relevant information.

$T\.code$ the actual sequence of instructions generated for the tree $T$.

$new\text{label}$ returns a $\text{label}$ to mark an instruction in the generated code which may be the $\text{target}$ of a jump.

$emit$ emits an instructions (regarded as a $\text{string}$).
IR: Infrastructure

Colour and font coding of IR code generation

- **Green**: Nodes of the Abstract Syntax Tree
- **Brown**: Characters and strings of the Intermediate Representation
- **Red**: Variables and data structures of the *language* in which the IR code generator is written
- **blue**: Names of relevant *procedures* used in IR code generation.
- **Black**: All other stuff.
IR: Example

\[
E \quad \rightarrow \quad \text{id} \quad \triangleright
\]

\[
E.place \quad := \quad \text{id}.place;
E.code \quad := \quad \text{emit}()
\]

\[
E_0 \quad \rightarrow \quad E_1 - E_2 \quad \triangleright
\]

\[
E_0.place \quad := \quad \text{newtemp};
E_0.code \quad := \quad E_1.code
\lll
\lll
emit(E_0.place := E_1.place - E_2.place)
\]
IR: Example

\[ S \rightarrow \text{id := } E \]

\[ S\.code \quad := \quad E\.code \]
\[ \quad || \quad \text{emit(id.place := E.place)} \]

\[ S_0 \rightarrow \text{while } E \text{ do } S_1 \]

\[ S_0\.begin \quad := \quad \text{newlabel;} \]
\[ S_0\.after \quad := \quad \text{newlabel;} \]
\[ S_0\.code \quad := \quad \text{emit(S_0.begin:)} \]
\[ \quad || \quad E\.code \]
\[ \quad || \quad \text{emit(if } E\.place = 0 \text{ goto } S_0\.after) \]
\[ \quad || \quad S_1\.code \]
\[ \quad || \quad \text{emit(goto } S_0\.begin) \]
\[ \quad || \quad \text{emit(S_0.after:)} \]
IR: Example

\[ S \rightarrow \text{id} := E \quad \triangleright \]

\[
S.\text{code} := E.\text{code} \\
|| \quad \text{emit} (\text{id} . \text{place} := E.\text{place})
\]

\[ S_0 \rightarrow \text{while} \ E \ \text{do} \ S_1 \quad \triangleright \]

\[
S_0.\text{begin} := \text{newlabel}; \\
S_0.\text{after} := \text{newlabel}; \\
S_0.\text{code} := \text{emit} (S_0.\text{begin}:) \\
|| \quad E.\text{code} \\
|| \quad \text{emit} (\text{if} \ E.\text{place} = 0 \ \text{goto} \ S_0.\text{after}) \\
|| \quad S_1.\text{code} \\
|| \quad \text{emit} (\text{goto} S_0.\text{begin}) \\
|| \quad \text{emit} (S_0.\text{after}:)
\]
IR: Generation

While generating the intermediate representation, it is sometimes necessary to generate jumps into code that has not been generated as yet (hence the address of the label is unknown). This usually happens while processing

- **forward** jumps
- **short-circuit** evaluation of boolean expressions

It is usual in such circumstances to either fill up the empty label entries in a **second pass** over the the code or through a process of **backpatching** (which is the maintenance of lists of jumps to the same instruction number), wherein the blank entries are filled in once the sequence number of the target instruction becomes known.
A Calling Chain

Main program
  Globals

Procedure P2
  Locals of P2
  Procedure P21
    Locals of P21
    Body of P21
    Call P21
  Body of P2
  Call P21

Procedure P1
  Locals of P1
  Body of P1
  Call P2

Main body
  Call P1
Run-time Structure: 1

- **Main program**
  - **Globals**
  - **Procedure P2**
    - Locals of P2
      - **Procedure P21**
        - Locals of P21
        - Body of P21
      - Body of P2
    - Body of P2
  - **Procedure P1**
    - Locals of P1
    - Body of P1
  - Main body
  - **Globals**
Run-time Structure: 2

Main program

Globals

Procedure P2
Locals of P2

Procedure P21
Locals of P21
Body of P21
Body of P2

Procedure P1
Locals of P1
Body of P1
Main body

Return address to Main
Dynamic link to Main
Locals of P1
Static link to Main
Formal par of P1

Globals
Run-time Structure: 3

Main program

Globals

Procedure P2
Locals of P2

Procedure P21
Locals of P21

Body of P2

Procedure P1
Locals of P1

Body of P1

Body of P21

Return address to last of P1
Dynamic link to last P1
Locals of P2
Static link to last P1
Formal par P2

Return address to Main
Dynamic link to Main
Locals of P1
Static link to Main
Formal par of P1

Globals
Run-time Structure: 4

Main program

Globals

Procedure P2
Locals of P2

Procedure P21
Locals of P21

Procedure P1
Locals of P1

Main body

Return address to last of P2
Dynamic link to last P2
Locals of P21
Static link last P2
Formal par P21

Return address to last of P1
Dynamic link to last P1
Locals of P2
Static link to last P1
Formal par P2

Return address to Main
Dynamic link to Main
Locals of P1
Static link to Main
Formal par of P1

Globals
Run-time Structure: 5

Main program
- Globals
- Procedure P2
  - Locals of P2
  - Procedure P21
    - Locals of P21
    - Body of P21
  - Body of P2
- Procedure P1
  - Locals of P1
  - Body of P1
- Main body

Return address to last of P21
- Dynamic link to last P21
- Locals of P21
  - Static link to last P2
  - Formal par P21

Return address to last of P2
- Dynamic link to last P2
- Locals of P21
  - Static link last P2
  - Formal par P21

Return address to last of P1
- Dynamic link to last P1
- Locals of P2
  - Static link to last P1
  - Formal par P2

Return address to Main
- Dynamic link to Main
- Locals of P1
  - Static link to Main
  - Formal par of P1

Globals