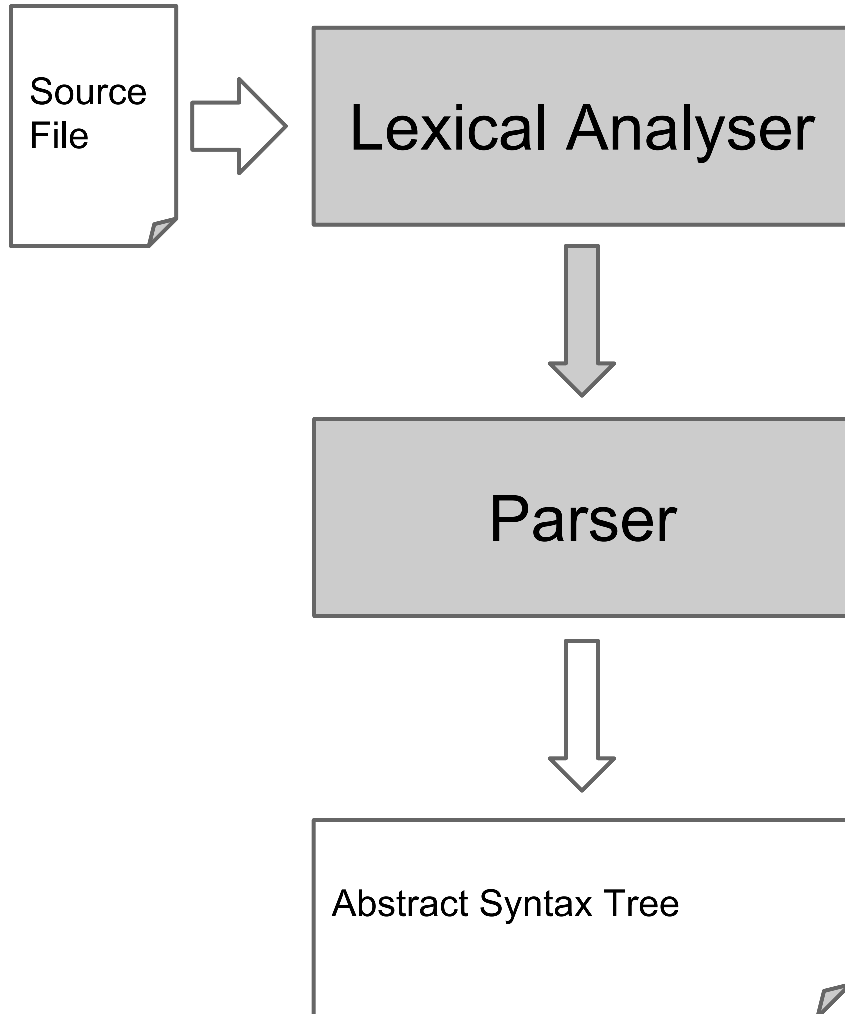
CPS2000
Compiler Theory & Practice

Notes on Handcrafting a Parser

Gordon Mangion

Compiler



Keyword Table

Abstract Syntax Tree

Symbol Table?

Error Module?

Parser

What is the actual input to the Parser?

What is the parser doing?

How do we implement it?

What is its output?

Parsing

- Syntax Analysis
  - Hierarchical Analysis
  - The input (Program) conforms to the grammar
  - Builds a data structure
    - Parse Tree
    - AST - Abstract Syntax Tree
  - Side-effect
    - Symbol-table entries?

We are considering a Top-down recursive descent parser

Lexer and Parser

Connecting the Lexer with the Parser options

1. Parser calls lex.NextToken()

2. TokenStream
   a. Read all file and generate Tokens
   b. Class acts as a stream of tokens
   c. Parser can move through the stream as it pleases

Grammar Example

<Letter> ::= [A-Za-z]

<Digit> ::= [0-9]

<**IntegerLiteral**> ::= <Digit> { <Digit> }

<**Identifier**> ::= (<Letter> | ‘_’) { <Letter> | “_” | <Digit> }

<**Expression**> ::= <**Identifier**> | <**IntegerLiteral**>

<**Assignment**> ::= ‘**set**’ <**Identifier**> ‘**<-**’ <**Expression**>

<**Statement**> ::= <**Assignment**> ‘**;**’ | <**Expression**> ‘**;**’

<***Program***> ::= { <**Statement**> }

Parser

1. Lexer
    a. Terminals
    b. Patterns

2. Parser
    a. Non-Terminals
        i. Create a procedure to parse its rule
        ii. Create an AST node to insert in the AST
        iii. Depending on the rule might need to read/write into the symbol-table
    b. Start with the Start Non-Terminal and follow the production-rules

Grammar Example

<Letter>	::= [A-Za-z]
<Digit>	::= [0-9]
<IntegerLiteral>	::= <Digit> { <Digit> }
<Identifier>	::= (<Letter> | ‘_’) { <Letter> | “_” | <Digit> }
<Expression>	::= <Identifier> | <IntegerLiteral>
<Assignment>	::= **‘set’** <Identifier> **‘<-’** <Expression>
<IfStatement>	::= ‘if’ ‘(’ <Expression> ‘)’ <Block> [‘else’ <Block>]
Block	::= ‘{‘ { <Statement> } ‘}’
<Statement>	::= <Assignment> **‘;’** | <IfStatement> **‘;’**
<***Program***>	::= { <Statement> }

‘if’ ‘(’ expr ‘)’ Block [‘else’ Block]

```
bool Parse_If( … )
{
    if( nextToken().Kind != LParen )
    {
        Error …
        return false;
    }
```

‘if’ ‘(’ expr ‘)’ Block [‘else’ Block]

```
if( !Parse_Expression(...) )
{
    Error …
    return false;
}
```

‘if’ ‘(’ expr ‘)’ Block [‘else’ Block]

```
if( nextToken().Kind != tk_RParen ) {
    Error …
    return false;
}

if( !Parse_Block(...) ) {
    Error …
    return false;
}
```

‘if’ ‘(’ expr ‘)’ Block [‘else’ Block]

```
if( nextToken().Kind != tk_Else ) {
    return true;
}
if( !Parse_Block(...) ) {
    Error …
    return false;
}
return true;
} // Parse_If
```

Grammar Example

<Letter> ::= [A-Za-z]

<Digit> ::= [0-9]

<IntegerLiteral> ::= <Digit> { <Digit> }

<Identifier> ::= (<Letter> | ‘_’) { <Letter> | “_” | <Digit> }

<Expression> ::= <Identifier> | <IntegerLiteral>

<Assignment> ::= **‘set’** <Identifier> **‘<-’** <Expression>

<IfStatement> ::= ‘if’ ‘(’ <Expression> ‘)’ <Block>
[‘else’ <Block>]

<Statement> ::= <Assignment> ‘;’ | <IfStatement> ‘;’

<***Program***> ::= { <Statement> }

<Statement> ::= <Assignment> ‘;’ | <IfStatement>

```
bool Parse_Statement(...)
{
    if( nextToken().Kind == tk_set )
    {
        if(!Parse_Assignment(...))
        { error … return false; }

        return (nextToken().Kind == tk_semicolon)
    }
    else if( nextToken().Kind == tk_if )
        return Parse_If(...);
}
```

<<u>Program</u>> ::= { <Statement> }

```
bool Parser::Parse(...)  {
    while( currentToken.Kind != tk_EOF )
    {
        if(!Parse_Statement(...))
        {
            Error … return false;
        }
    }
    return true; }
```

Abstract Syntax Tree

ASTNode

Data

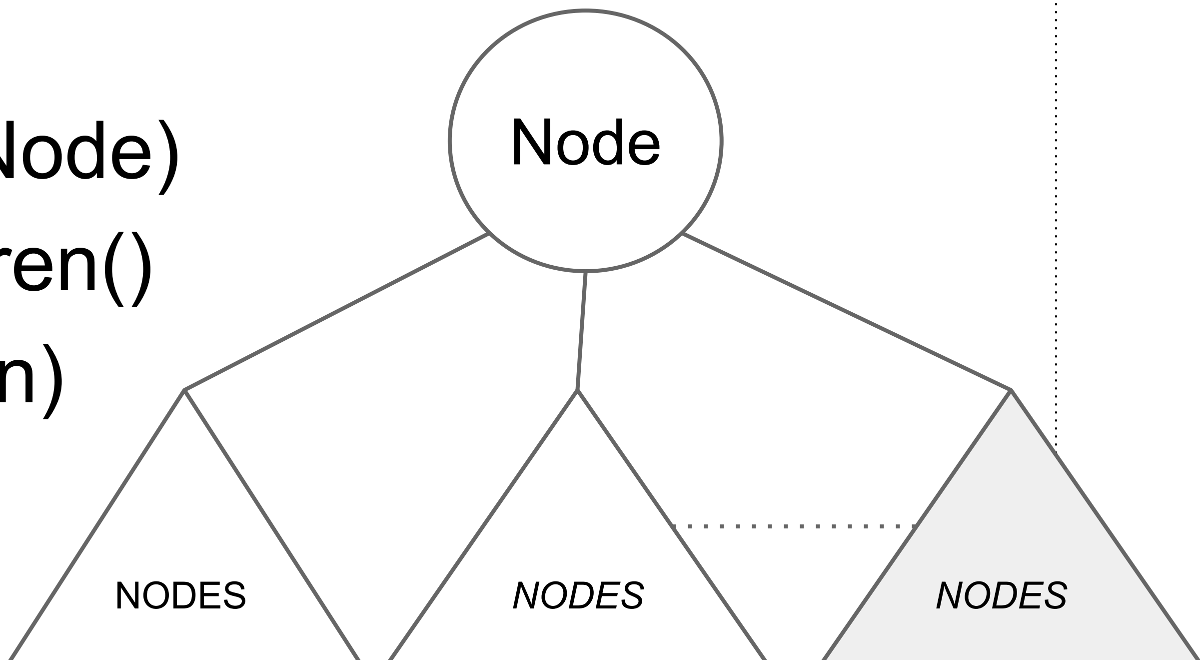
List of Children (Node)

Operations

void AddChild(Node)

int NumOfChildren()

Node GetChild(n)



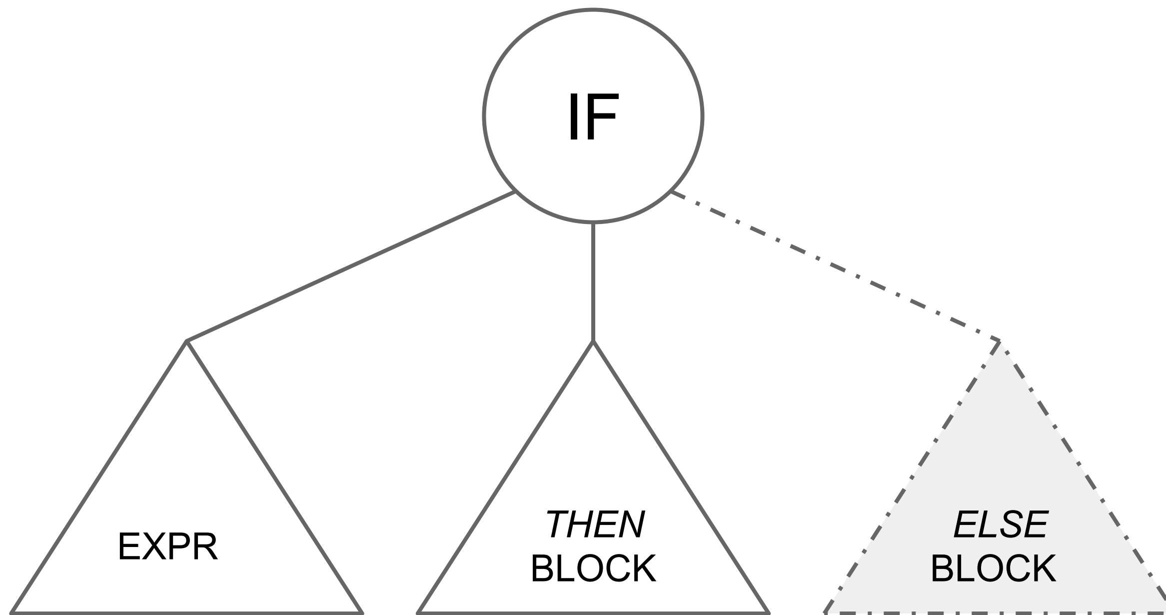
Parse Method

```
ASTNode Parser::Parse_If(....)
{
    ASTNode *pNode = new IfNode()
    …

    return pNode;
}
```

Abstract Syntax Tree

‘if’ ‘(’ expr ‘)’ Block [‘else’ Block]

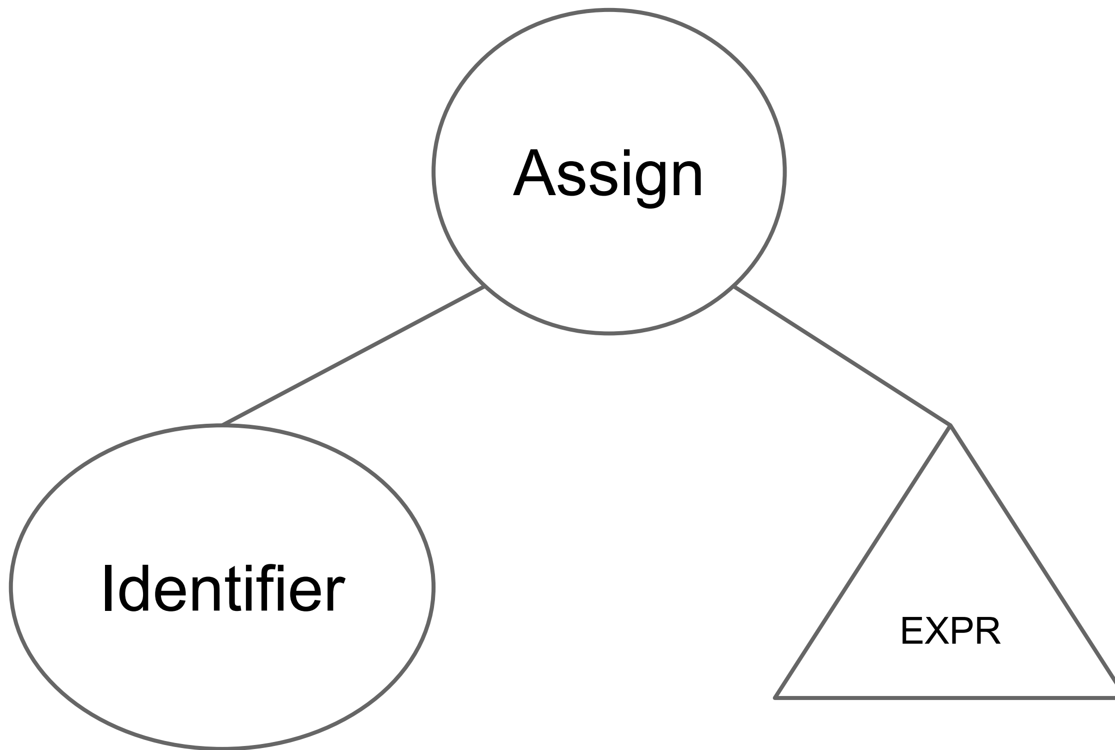


Creating a sub-tree example

```
ASTNode Parser::Parse_If(....)
{   ...
    ASTNode *pIfNode = new IfNode()
    pIfNode->AddChild( Parse_Expr(...) )
    pIfNode->AddChild( Parse_Block(...) )
    if(....)
        pIfNode->AddChild( Parse_Block(...) )
    return pNode;
    …
}
```

Abstract Syntax Tree

‘set’ <identifier> ‘<-’ <expr>



Creating a sub-tree example

```
ASTNode Parser::Parse_Assign(....)
{
    ASTNode *pAsgnNode = new AssignNode()

    pAsgnNode->AddChild( Parse_Ident(...) )
    pAsgnNode->AddChild( Parse_Expr(...) )

    return pNode;
}
```

Traversing the AST

Generic Traverse Algorithm

```
    Do Something with current Node

    Foreach child
        Do Something with current Node
        call Traverse on child
        Do Something with current Node

    Do Something with current Node
```

Traversing the AST

Stages traverse the AST

- Semantic Analysis / Type Checker

- Backend Stages
  - Optimisation
  - Code Generation
  - Interpreters / Execution

Interface (Java)

```
public interface IAdder
{
    int add(int n, int m);
}


public interface IMinus
{
    int subtract(int n, int m);
}
```

Interface (Java)

```
public class SimpleMath
        implements IAdder, IMinus
{
    public int add(int n, int m)
    { return n+m; }

    public int subtract(int n, int m)
    { return n-m; }
}
```

Interface (Java)

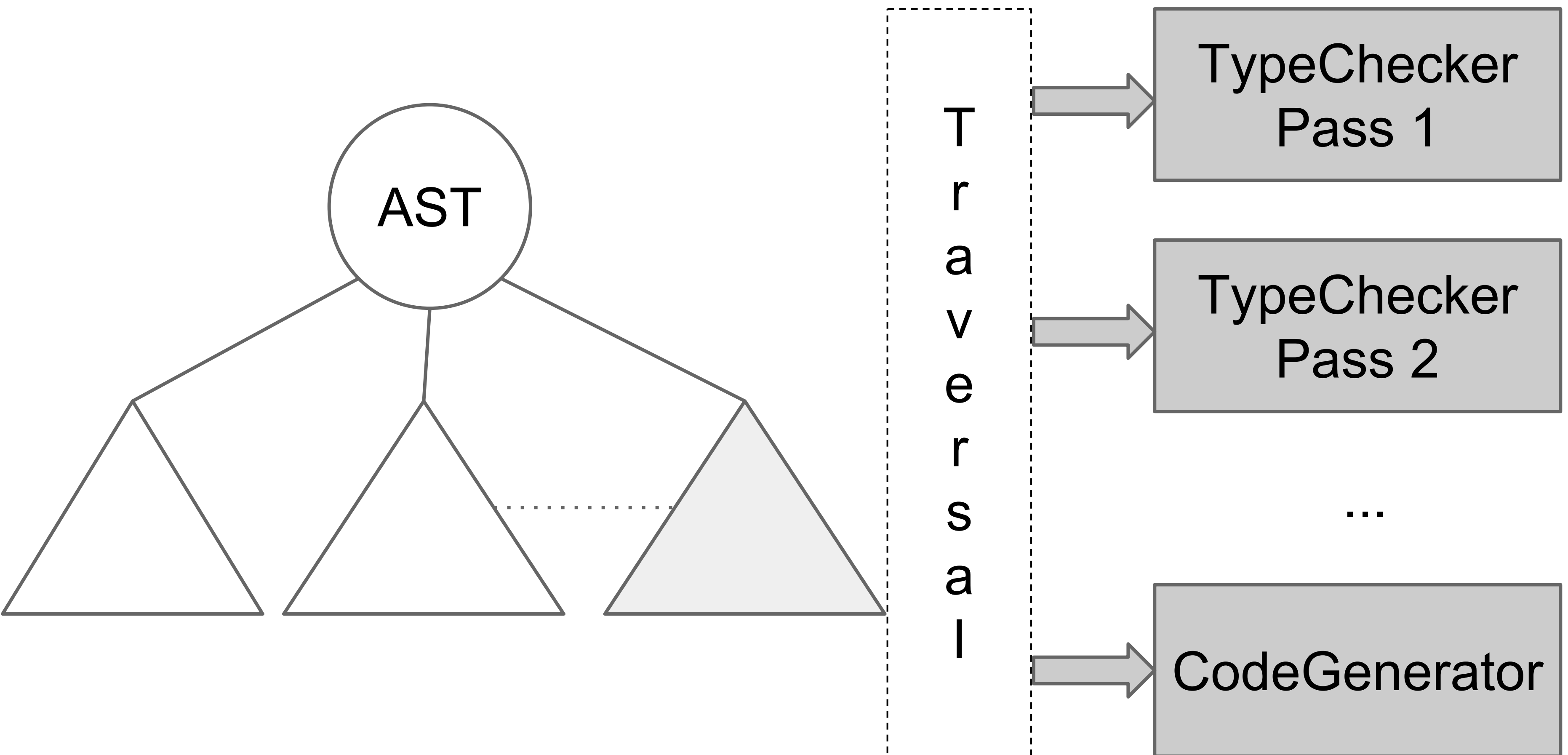
```
public static void main(…)
{
    SimpleMath sm = new SimpleMath()
    IAdder a = sm;
    IMinus s = sm;
    a.add(1,2);
    s.subtract(4,2);
}
```

Interface (C++)

```
abstract class IAdder
{
    int add(int n, int m) = 0;
};


abstract class IMinus
{
    int subtract(int n, int m) = 0;
};
```

AST Traversal



Visitor Design Pattern

ASTNode
    Traverse()
        Traverse Algo

IfNode
    Traverse()
        Traverse Algo

Visitor Design Pattern

ASTNode
    Accept()

IfNode
    Accept()

Visitor Class

Visit(ASTNode *pNode)
    Traverse Algorithm

Visit(IfNode *pNode)
    Traverse Algorithm

Visitor Design Pattern

ASTNode
    Accept(Visitor v)
        v.visit(this);

IfNode
    Accept(Visitor v)
        v.visit(this)

Visitor Class

Visit(ASTNode *pNode)
    Traverse Algorithm

Visit(IfNode *pNode)
    Traverse Algorithm

Visitor Design Pattern

ASTNode

    Accept(IVisitor v)

        v.visit(this);

Visitor Class : IVisitor

Visit(ASTNode *pNode)

    Traverse Algorithm

Interface IVisitor

Visit(ASTNode *pNode);

SymbolTable

Set of Symbols (definitions)

Tracks current bindings of identifiers

Stack of symbols:

Push(x)	Add a symbol x to table
Pop	Remove top symbol
Find(x)	Find a symbol x in the table starting from the topmost entry

SymbolTable

Stack of scopes:

enter_scope()	Start a new scope
find_symbol(x)	Find current x (or null)
add_symbol(x)	Add a symbol x to table
check_scope(x)	true if x defined in current scope
exit_scope()	Exit current scope