Compilers and Automata Theory

Joseph Thomas

University of Arizona
Department of Mathematics

November 14, 2011

What is a compiler?

What is a compiler?

- A **compiler** is a program that translates source code into some kind of executable machine code.

What is a compiler?

- A **compiler** is a program that translates source code into some kind of executable machine code.
- Mathematically, a mapping from programs written in one language (C, Java, Python, etc.) to programs written in another (x86, MIPS, Java Byte-Code, etc.).

What is a compiler?

- A **compiler** is a program that translates source code into some kind of executable machine code.
- Mathematically, a mapping from programs written in one language (C, Java, Python, etc.) to programs written in another (x86, MIPS, Java Byte-Code, etc.).
- Problem: How can I write a compiler that:

What is a compiler?

- A **compiler** is a program that translates source code into some kind of executable machine code.
- Mathematically, a mapping from programs written in one language (C, Java, Python, etc.) to programs written in another (x86, MIPS, Java Byte-Code, etc.).
- Problem: How can I write a compiler that:

What is a compiler?

- A **compiler** is a program that translates source code into some kind of executable machine code.
- Mathematically, a mapping from programs written in one language (C, Java, Python, etc.) to programs written in another (x86, MIPS, Java Byte-Code, etc.).
- Problem: How can I write a compiler that:
  - Is correct.

What is a compiler?

- A **compiler** is a program that translates source code into some kind of executable machine code.
- Mathematically, a mapping from programs written in one language (C, Java, Python, etc.) to programs written in another (x86, MIPS, Java Byte-Code, etc.).
- Problem: How can I write a compiler that:
  - Is correct.
  - Produces useful error messages.

What is a compiler?

- A **compiler** is a program that translates source code into some kind of executable machine code.
- Mathematically, a mapping from programs written in one language (C, Java, Python, etc.) to programs written in another (x86, MIPS, Java Byte-Code, etc.).
- Problem: How can I write a compiler that:
  - Is correct.
  - Produces useful error messages.
  - Produces efficient code.

The Automata Viewpoint

The Automata Viewpoint

- A **alphabet** is a set of symbols. (Ex: $\Sigma = \{\mathtt{a}, \mathtt{b}\}$)

The Automata Viewpoint

- A **alphabet** is a set of symbols. (Ex: $\Sigma = \{\mathtt{a}, \mathtt{b}\}$)
- **Strings** are ordered, finite tuples with entries in an alphabet. (Ex: $\mathtt{abaabba}$)

The Automata Viewpoint

- A **alphabet** is a set of symbols. (Ex: $\Sigma = \{\mathtt{a}, \mathtt{b}\}$)
- **Strings** are ordered, finite tuples with entries in an alphabet. (Ex: $\mathtt{abaabba}$)
- **Languages** are sets of strings. (Ex: $\{\mathtt{a}^n\mathtt{b}^n : n \in \mathbb{N}\}$)

The Automata Viewpoint

- A **alphabet** is a set of symbols. (Ex: $\Sigma = \{\mathtt{a}, \mathtt{b}\}$)
- **Strings** are ordered, finite tuples with entries in an alphabet. (Ex: $\mathtt{abaabba}$)
- **Languages** are sets of strings. (Ex: $\{\mathtt{a}^n\mathtt{b}^n : n \in \mathbb{N}\}$)
- Problem: Can I write down instructions that describe how to recognize when a string s is a member of language S?

The Automata Viewpoint

- A **alphabet** is a set of symbols. (Ex: $\Sigma = \{\mathtt{a}, \mathtt{b}\}$)
- **Strings** are ordered, finite tuples with entries in an alphabet. (Ex: $\mathtt{abaabba}$)
- **Languages** are sets of strings. (Ex: $\{\mathtt{a}^n\mathtt{b}^n : n \in \mathbb{N}\}$)
- Problem: Can I write down instructions that describe how to recognize when a string s is a member of language S?
- Problem: Can one method of "writing down instructions" be more powerful than another?

Who needs this technology?

Who needs this technology?

Anyone who needs to translate to or from a structured language.

Who needs this technology?

Anyone who needs to translate to or from a structured language.

- Programming

Who needs this technology?

Anyone who needs to translate to or from a structured language.

- Programming
- LaTeX, Web Browsers

Who needs this technology?

Anyone who needs to translate to or from a structured language.

- Programming
- LaTeX, Web Browsers
- Computer Algebra Systems (GAP, Matlab, Mathematica)

Who needs this technology?

Anyone who needs to translate to or from a structured language.

- Programming
- LaTeX, Web Browsers
- Computer Algebra Systems (GAP, Matlab, Mathematica)

Who needs this technology?

Anyone who needs to translate to or from a structured language.

- Programming
- LaTeX, Web Browsers
- Computer Algebra Systems (GAP, Matlab, Mathematica)

This talk will help you understand what these tools are doing...

Who needs this technology?

Anyone who needs to translate to or from a structured language.

- Programming
- LaTeX, Web Browsers
- Computer Algebra Systems (GAP, Matlab, Mathematica)

This talk will help you understand what these tools are doing...

- ...when they succeed in producing output.

Who needs this technology?

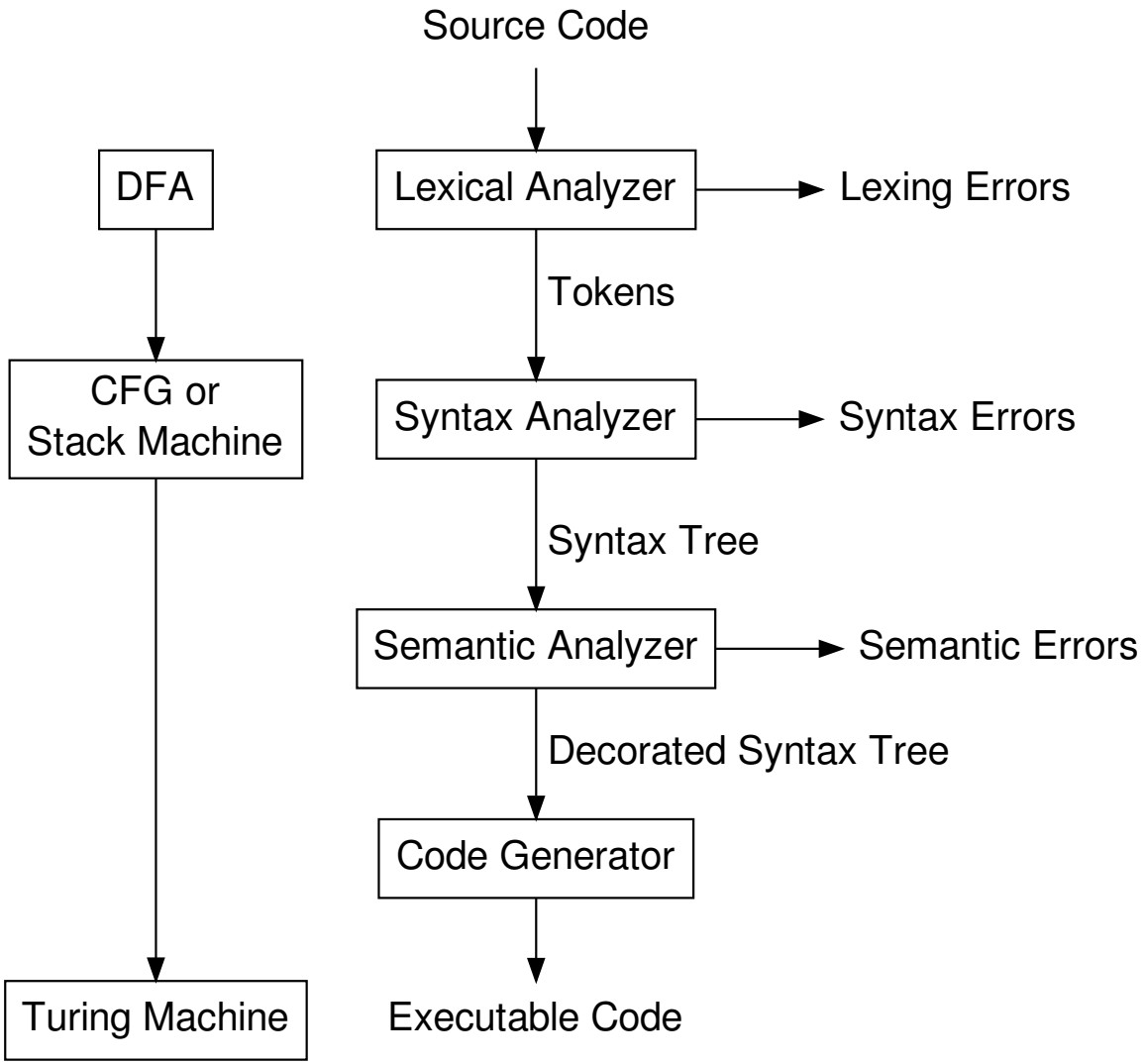
Anyone who needs to translate to or from a structured language.

- Programming
- LaTeX, Web Browsers
- Computer Algebra Systems (GAP, Matlab, Mathematica)

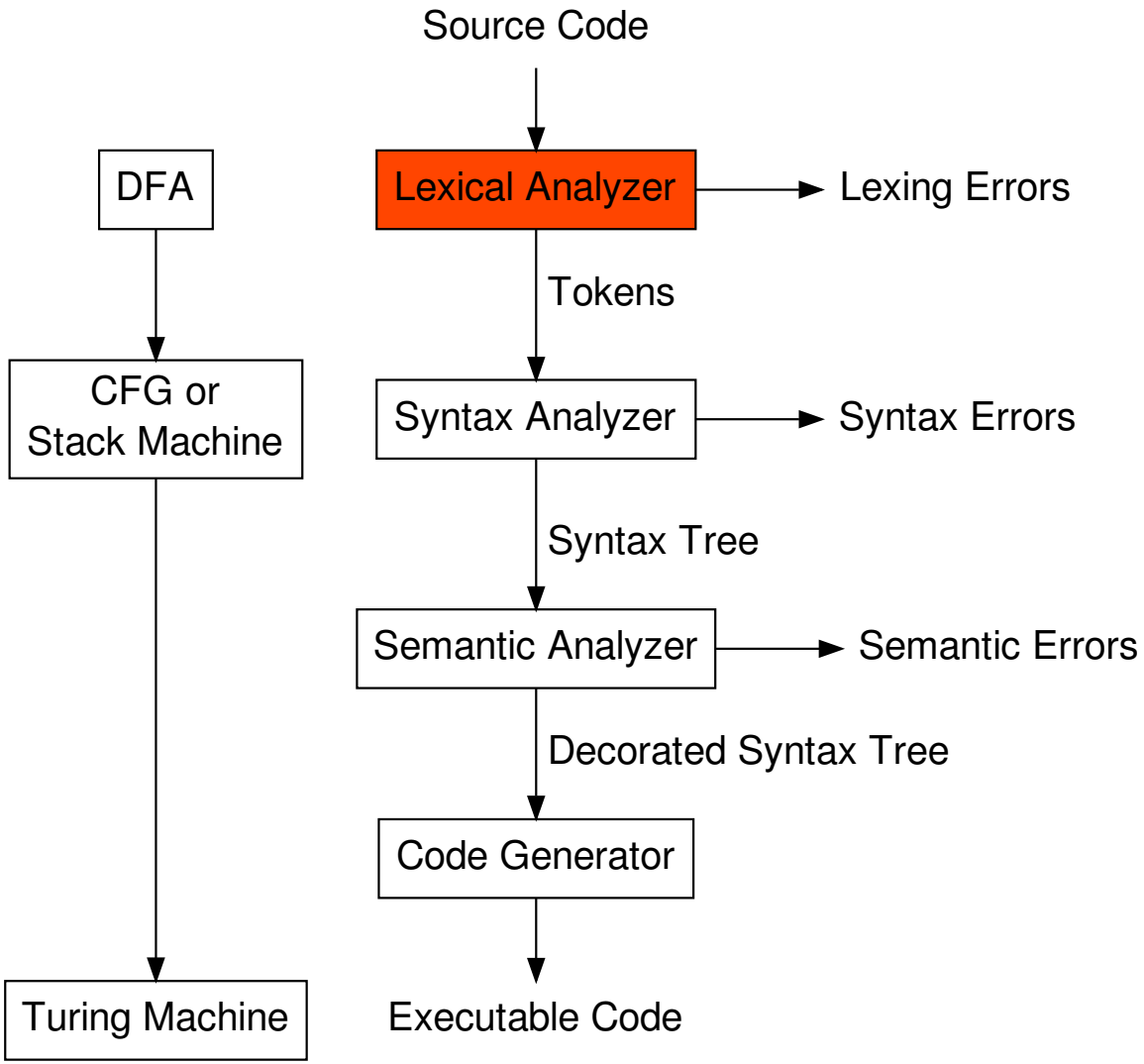
This talk will help you understand what these tools are doing...

- ...when they succeed in producing output.
- ...when they give you error messages.

Phases of Compilation



Lexical Analysis



Lexical Analysis

Main idea: Decompose a file (a string of characters) into "atomic" pieces.

Lexical Analysis

Main idea: Decompose a file (a string of characters) into "atomic" pieces.

- Recognize **reserved words** and **operators**:

Lexical Analysis

Main idea: Decompose a file (a string of characters) into "atomic" pieces.

- Recognize **reserved words** and **operators**:

Lexical Analysis

Main idea: Decompose a file (a string of characters) into "atomic" pieces.

- Recognize **reserved words** and **operators**:
  - `for` $\mapsto$ `FOR_BEGIN`

Lexical Analysis

Main idea: Decompose a file (a string of characters) into "atomic" pieces.

- Recognize **reserved words** and **operators**:
  - `for` $\mapsto$ `FOR_BEGIN`
  - `<=` $\mapsto$ `LESS_THAN_EQUAL`

Lexical Analysis

Main idea: Decompose a file (a string of characters) into "atomic" pieces.

- Recognize **reserved words** and **operators**:
  - `for` $\mapsto$ `FOR_BEGIN`
  - `<=` $\mapsto$ `LESS_THAN_EQUAL`
- Recognize numerical literals: `453` $\mapsto$ `INT_LIT(453)`.

Lexical Analysis

Main idea: Decompose a file (a string of characters) into "atomic" pieces.

- Recognize **reserved words** and **operators**:
  - `for` $\mapsto$ `FOR_BEGIN`
  - `<=` $\mapsto$ `LESS_THAN_EQUAL`
- Recognize numerical literals: `453` $\mapsto$ `INT_LIT(453)`.
- Recognize "spelling" errors:

Lexical Analysis

Main idea: Decompose a file (a string of characters) into "atomic" pieces.

- Recognize **reserved words** and **operators**:
  - `for` $\mapsto$ `FOR_BEGIN`
  - `<=` $\mapsto$ `LESS_THAN_EQUAL`
- Recognize numerical literals: `453` $\mapsto$ `INT_LIT(453)`.
- Recognize "spelling" errors:

Lexical Analysis

Main idea: Decompose a file (a string of characters) into "atomic" pieces.

- Recognize **reserved words** and **operators**:
  - `for` $\mapsto$ `FOR_BEGIN`
  - `<=` $\mapsto$ `LESS_THAN_EQUAL`
- Recognize numerical literals: `453` $\mapsto$ `INT_LIT(453)`.
- Recognize "spelling" errors:
  - `3.14.14` doesn't define a valid floating point number.

Lexical Analysis

Main idea: Decompose a file (a string of characters) into "atomic" pieces.

- Recognize **reserved words** and **operators**:
  - `for` $\mapsto$ `FOR_BEGIN`
  - `<=` $\mapsto$ `LESS_THAN_EQUAL`
- Recognize numerical literals: `453` $\mapsto$ `INT_LIT(453)`.
- Recognize "spelling" errors:
  - `3.14.14` doesn't define a valid floating point number.
  - `314!!foo` is not a valid variable name in C.

Example

A line of C code (30 characters with whitespace):

```
int foo = bar + 453;
```

Example

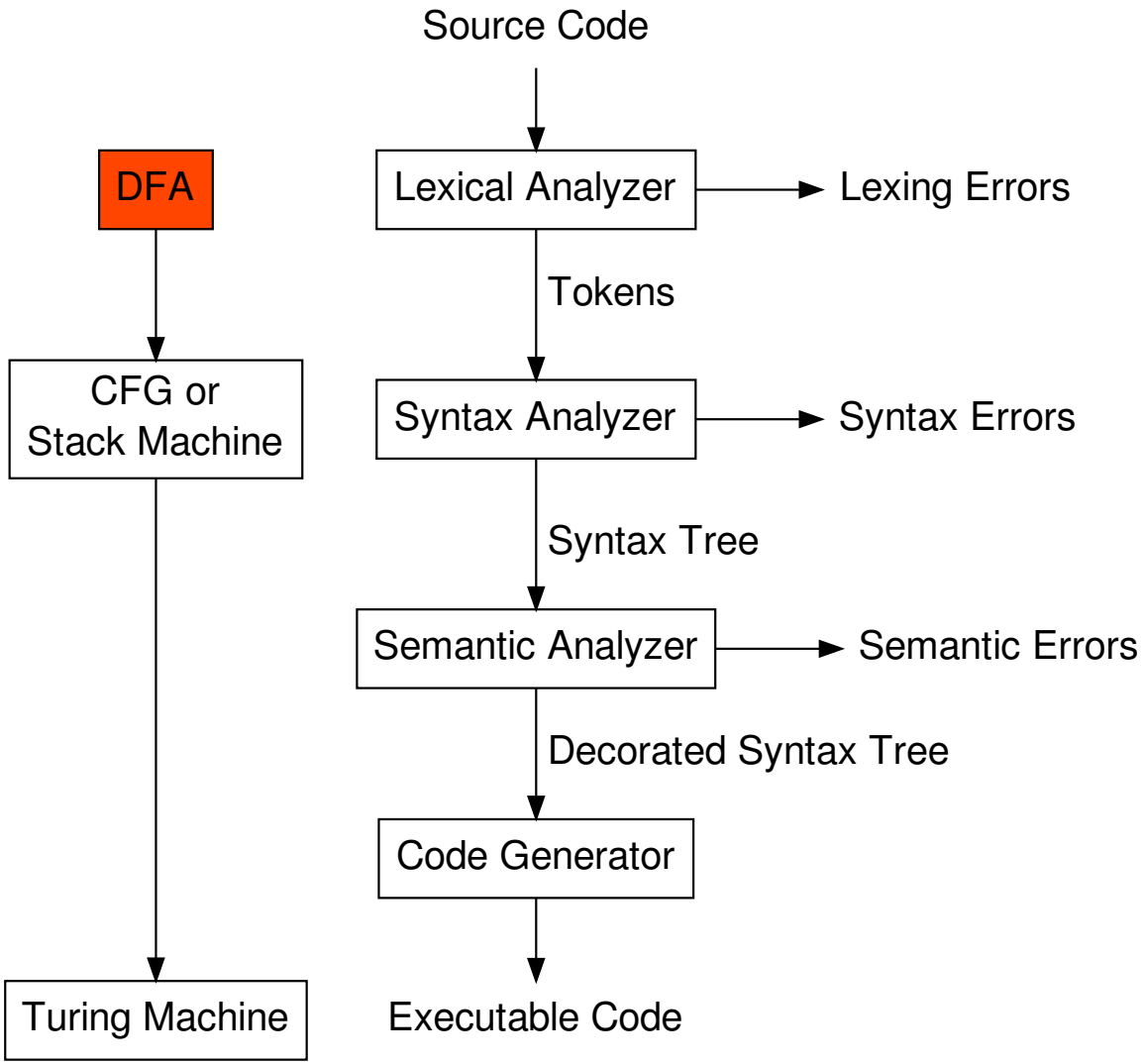
A line of C code (30 characters with whitespace):

```
int foo = bar + 453;
```

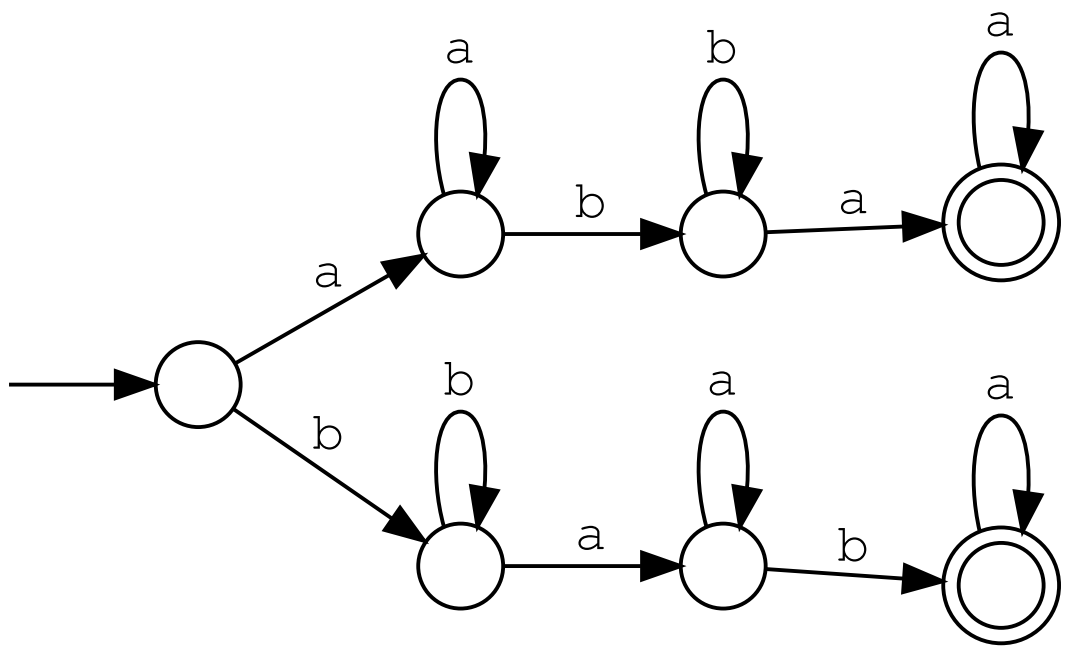
becomes a sequence of 7 tokens:

```
IDENT(int), IDENT(foo), EQUALS, IDENT(bar), PLUS,
INTLIT(453), SEMICOLON
```

Deterministic Finite Automatons (DFAs)

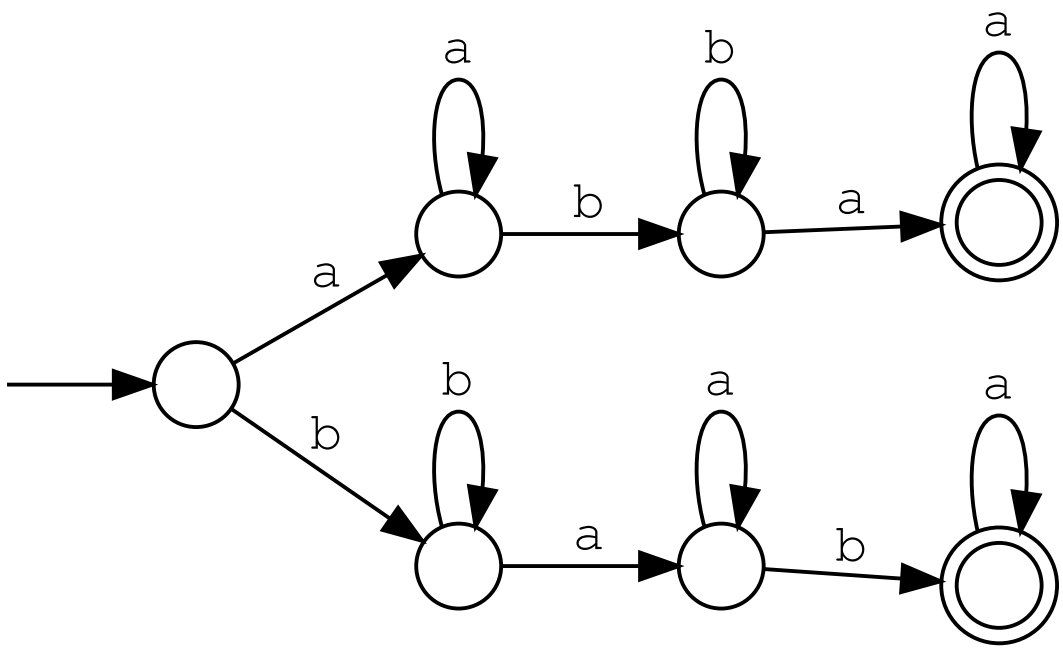


Deterministic Finite Automatons (DFAs)



DFAs are decorated directed graphs.

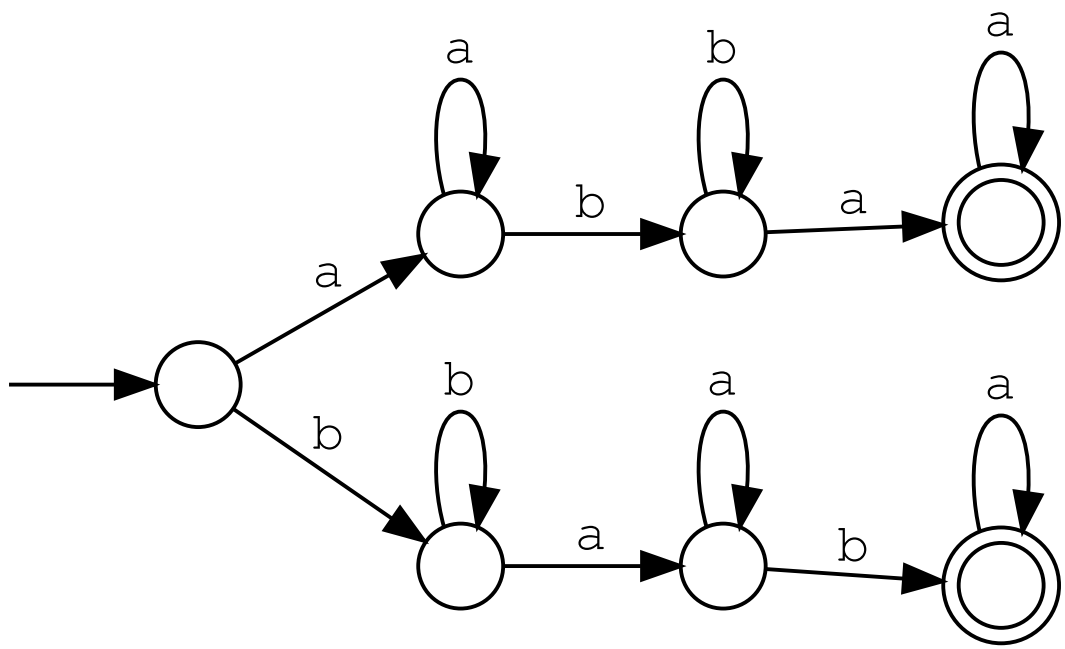
Deterministic Finite Automatons (DFAs)



DFAs are decorated directed graphs.

- Each edge identified with exactly one letter in the alphabet.

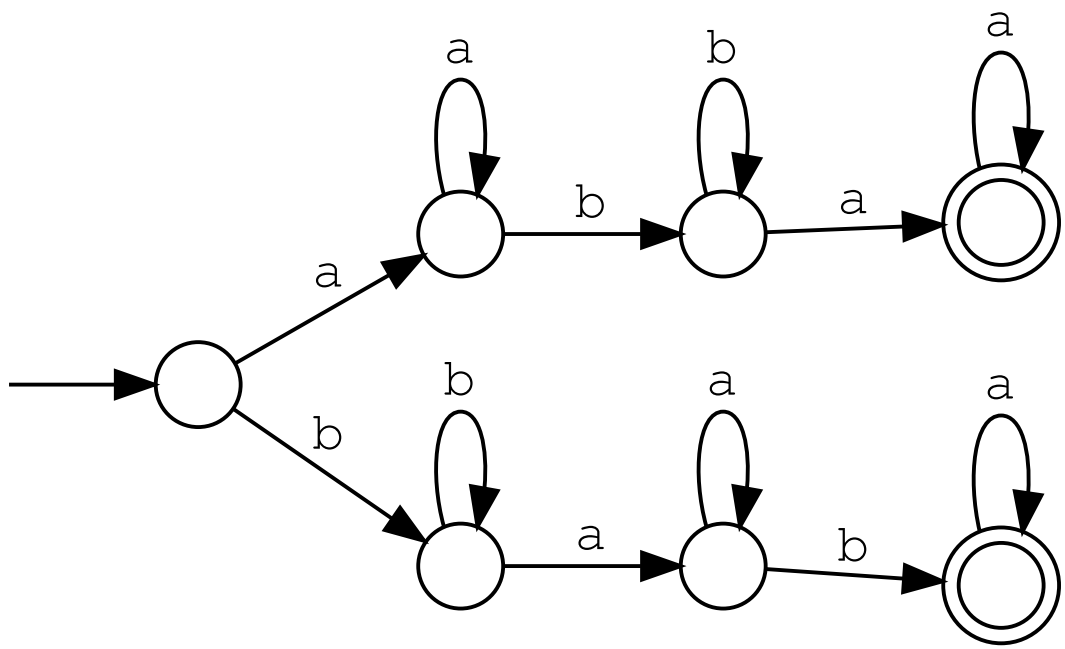
Deterministic Finite Automatons (DFAs)



DFAs are decorated directed graphs.

- Each edge identified with exactly one letter in the alphabet.
- One "start node," at least one "end node."

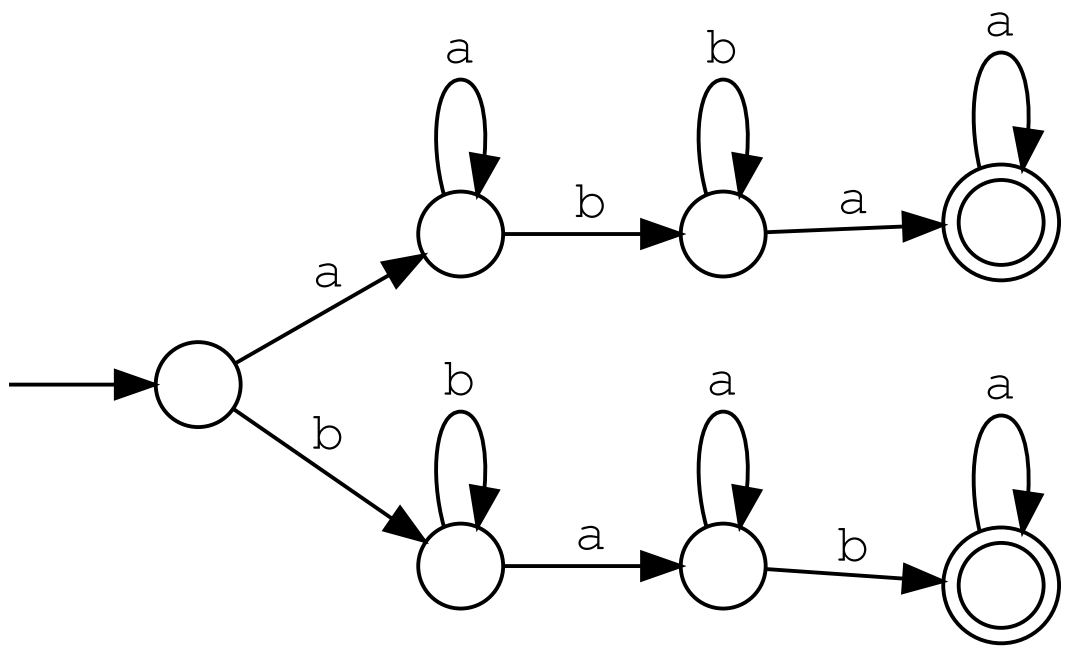
Deterministic Finite Automatons (DFAs)



DFAs are decorated directed graphs.

- Each edge identified with exactly one letter in the alphabet.
- One "start node," at least one "end node."

Deterministic Finite Automatons (DFAs)



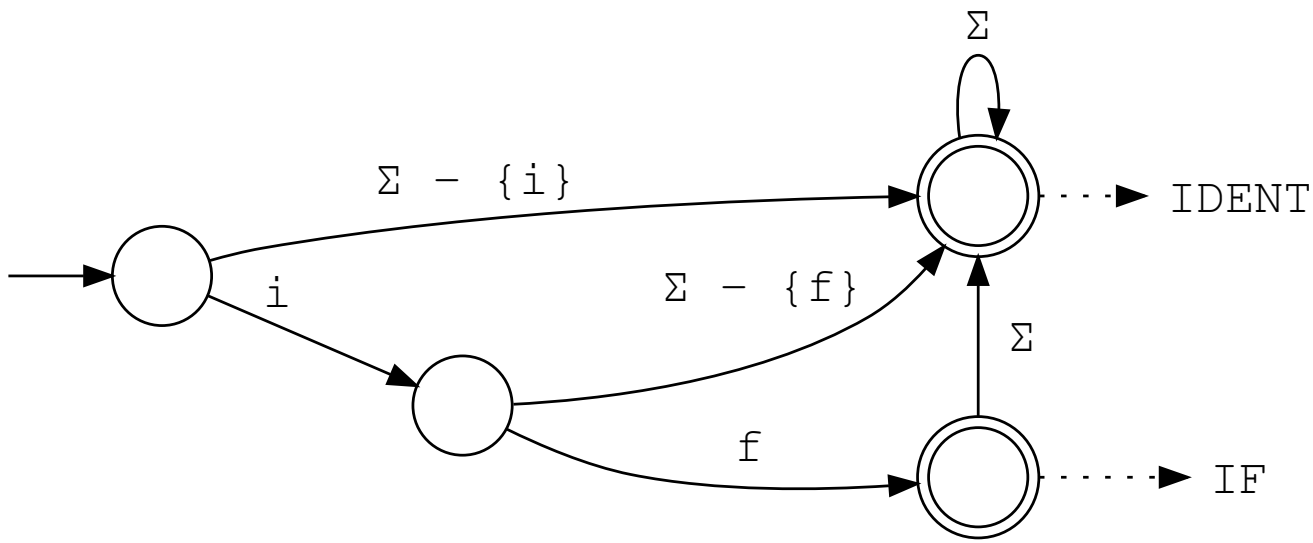
DFAs are decorated directed graphs.

- Each edge identified with exactly one letter in the alphabet.
- One "start node," at least one "end node."

A string is **recognized** (or **accepted**) by a given DFA if it describes a path from the start node to an end node.

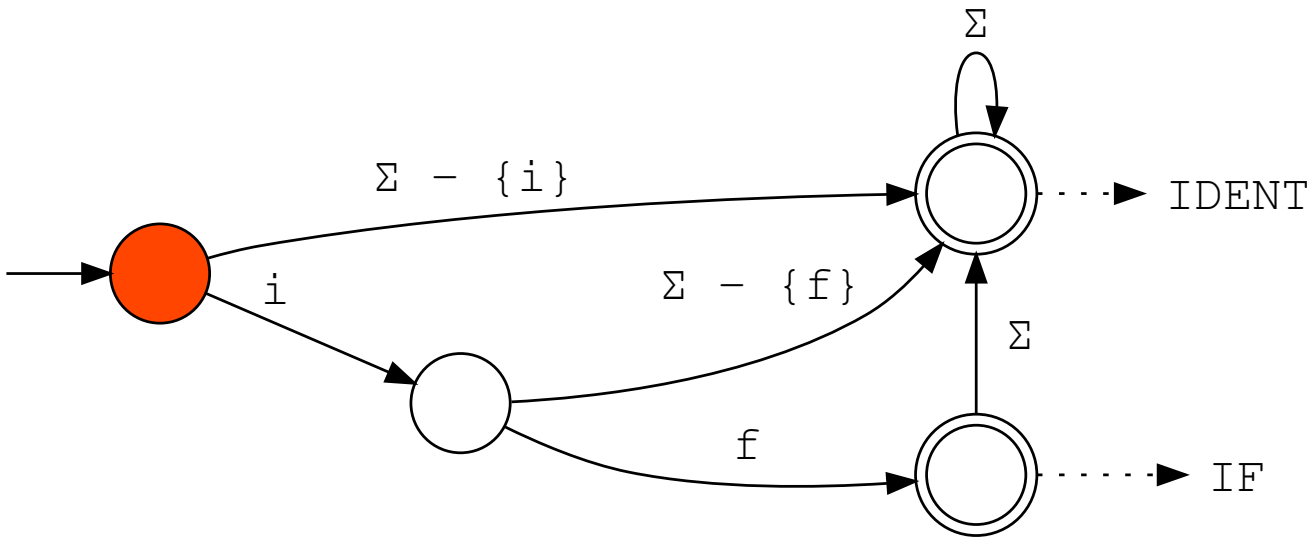
DFA Example

A DFA that recognizes identifiers (variable names) and the `if` token.



DFA Example

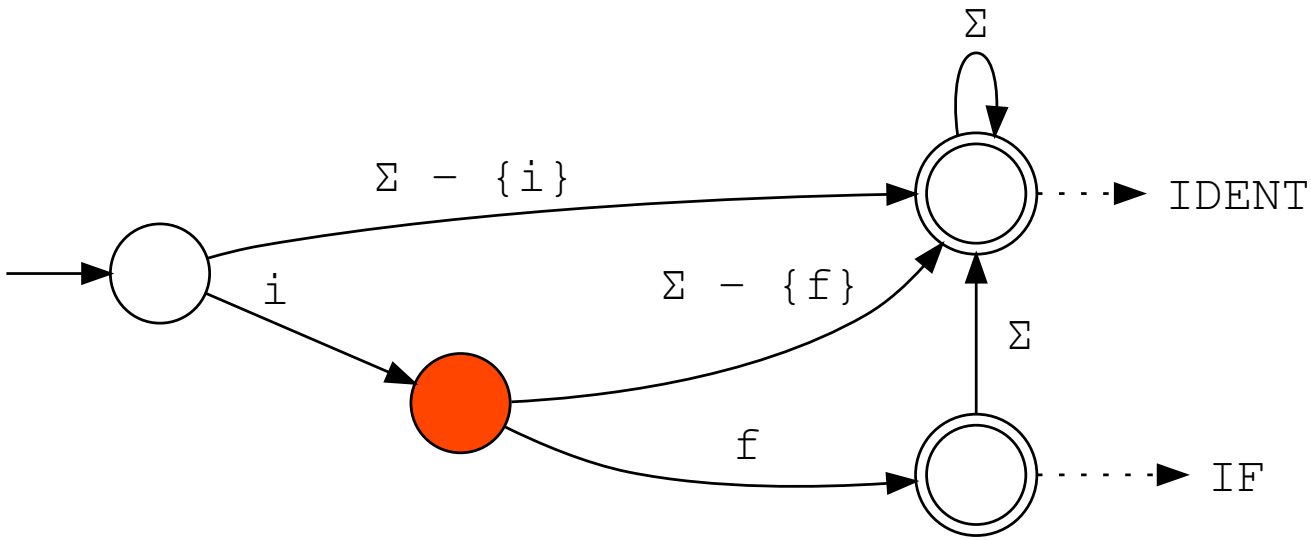
Input: `if iota`



Output:

DFA Example

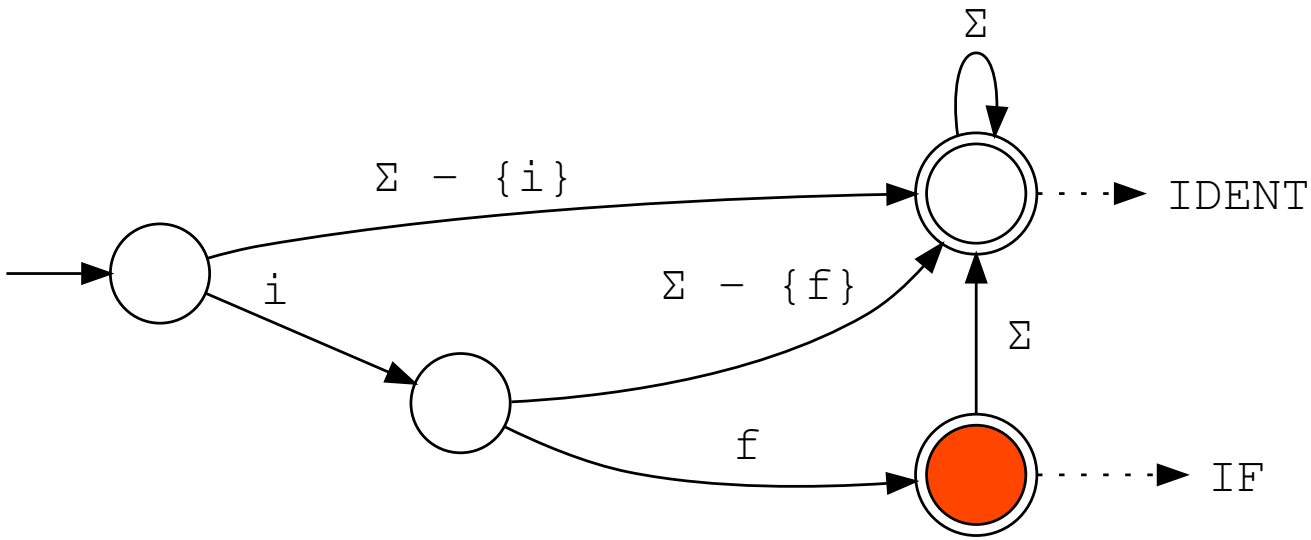
Input: `if iota`



Output:

DFA Example

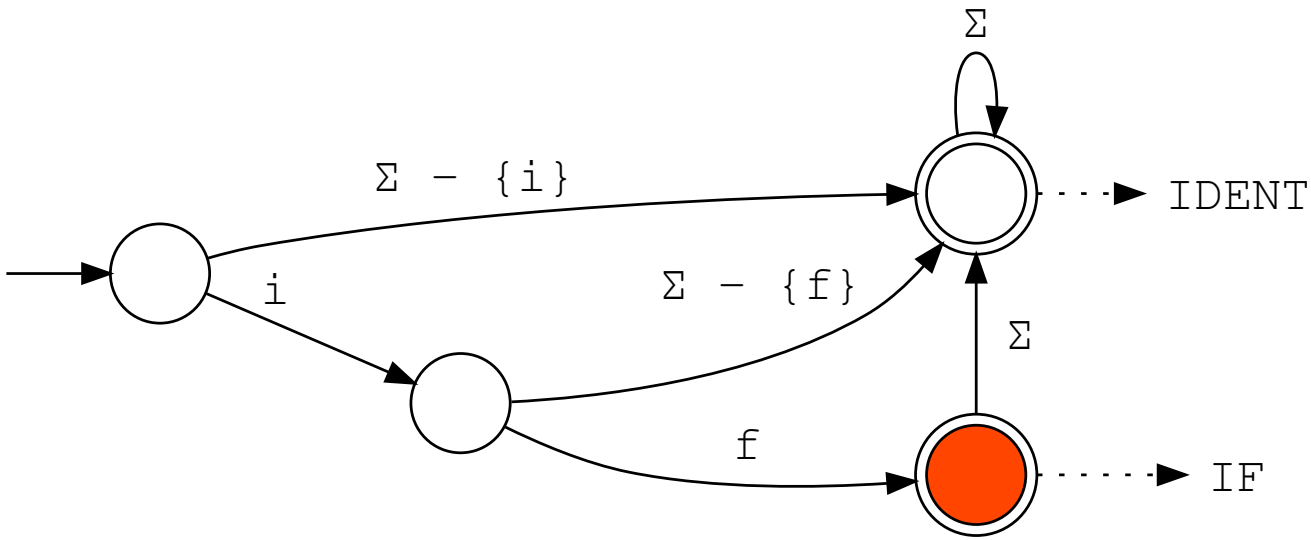
Input: `if iota`



Output:

DFA Example

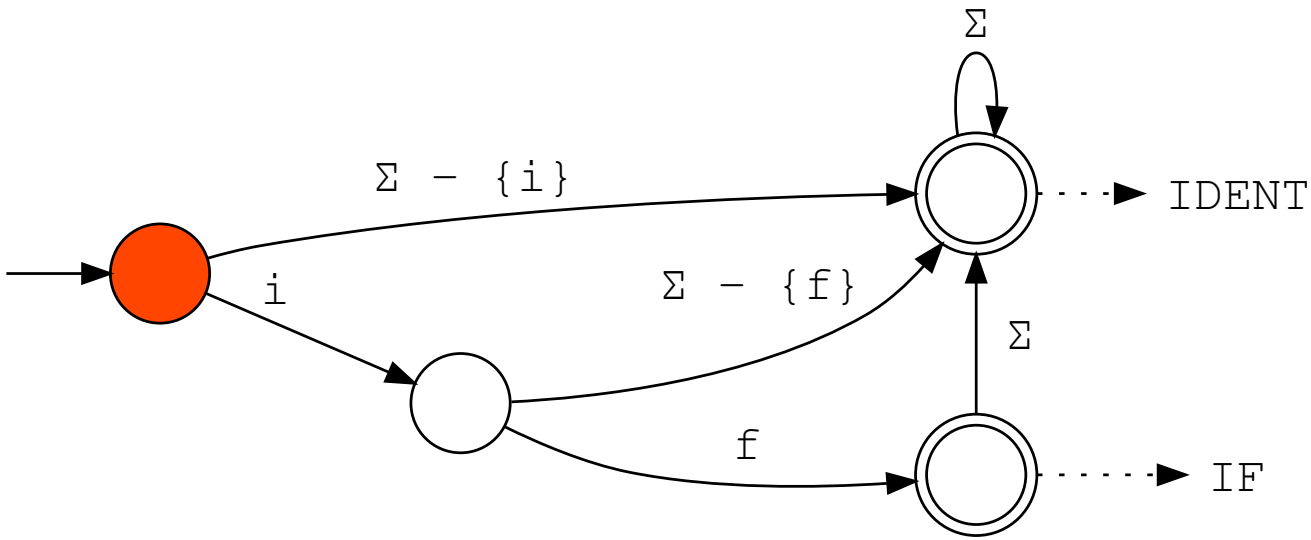
Input: `if iota`



Output: `IF`,

DFA Example

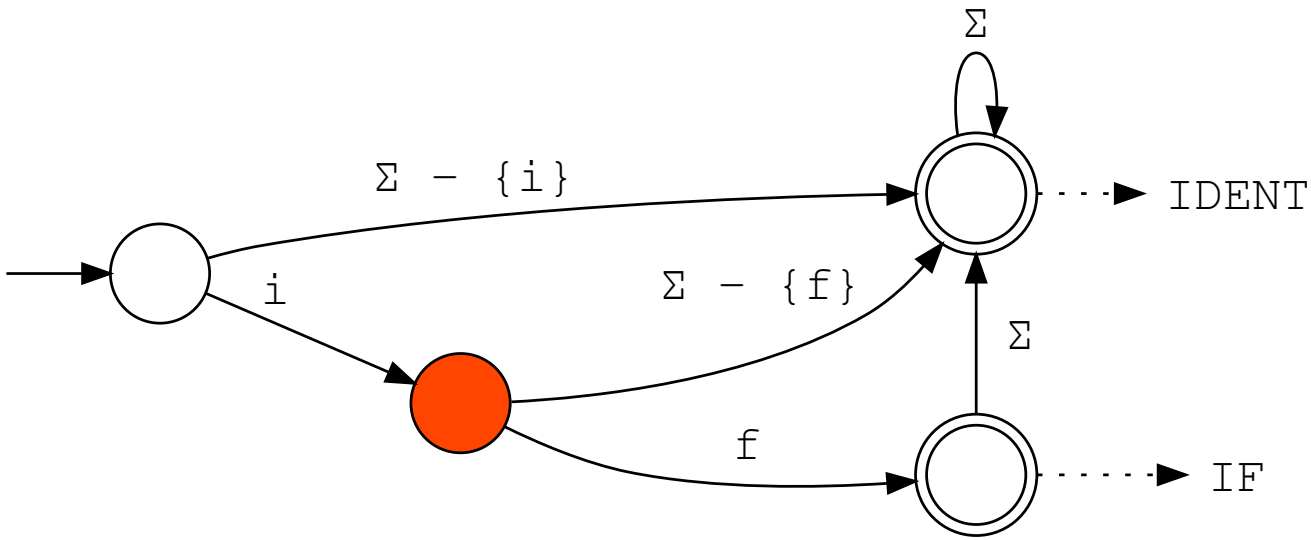
Input: `if iota`



Output: `IF`,

DFA Example

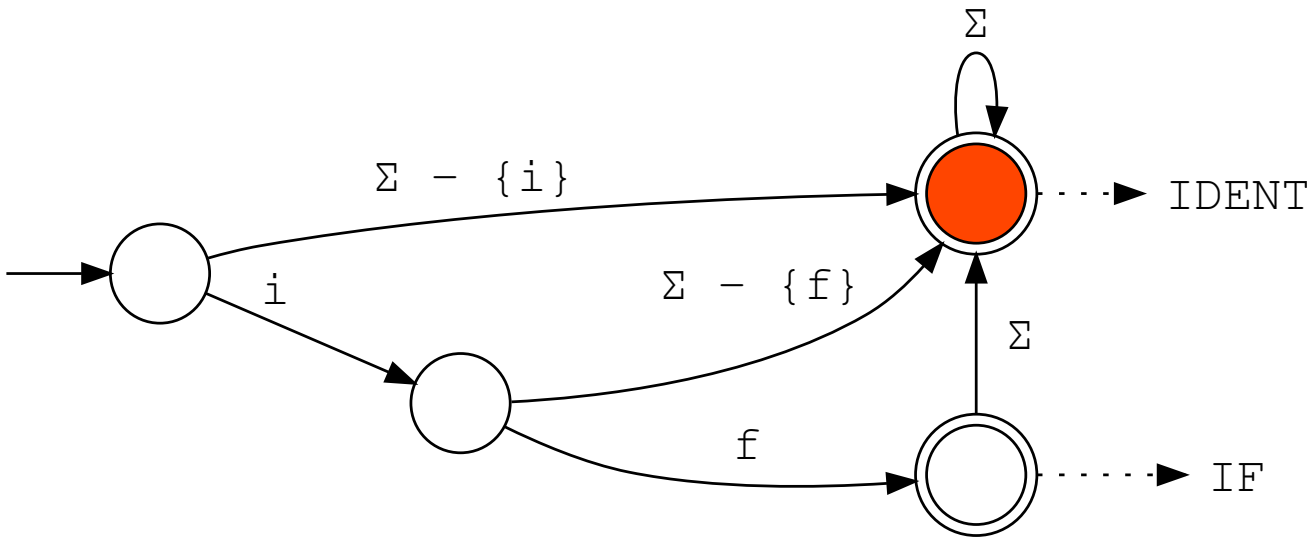
Input: `if iota`



Output: `IF`,

DFA Example

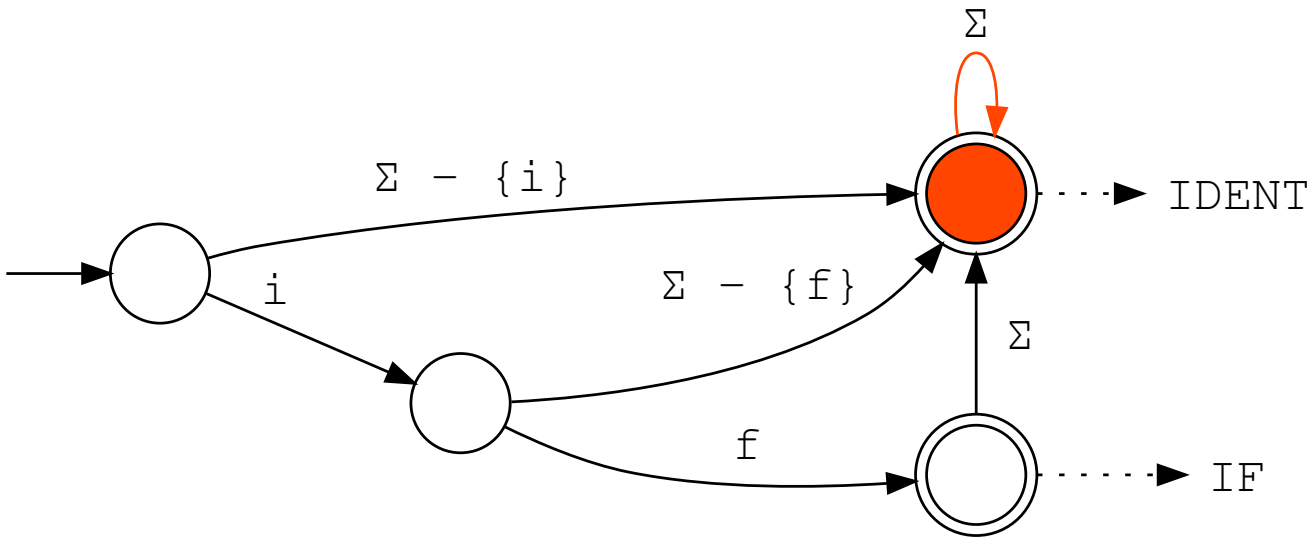
Input: `if iota`



Output: `IF`,

DFA Example

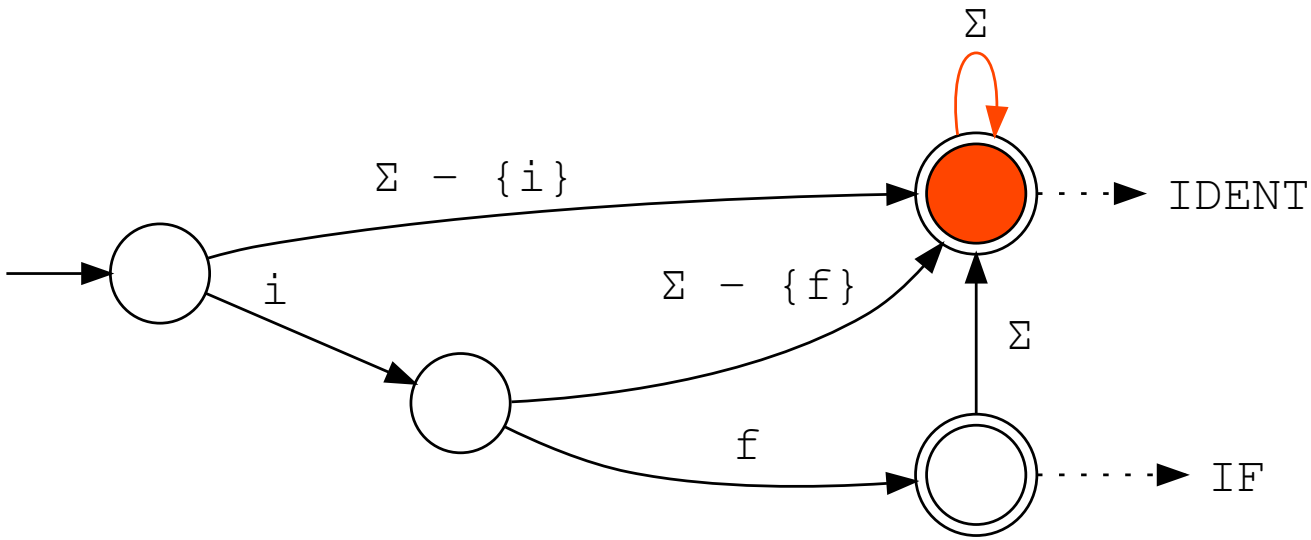
Input: `if iota`



Output: `IF`,

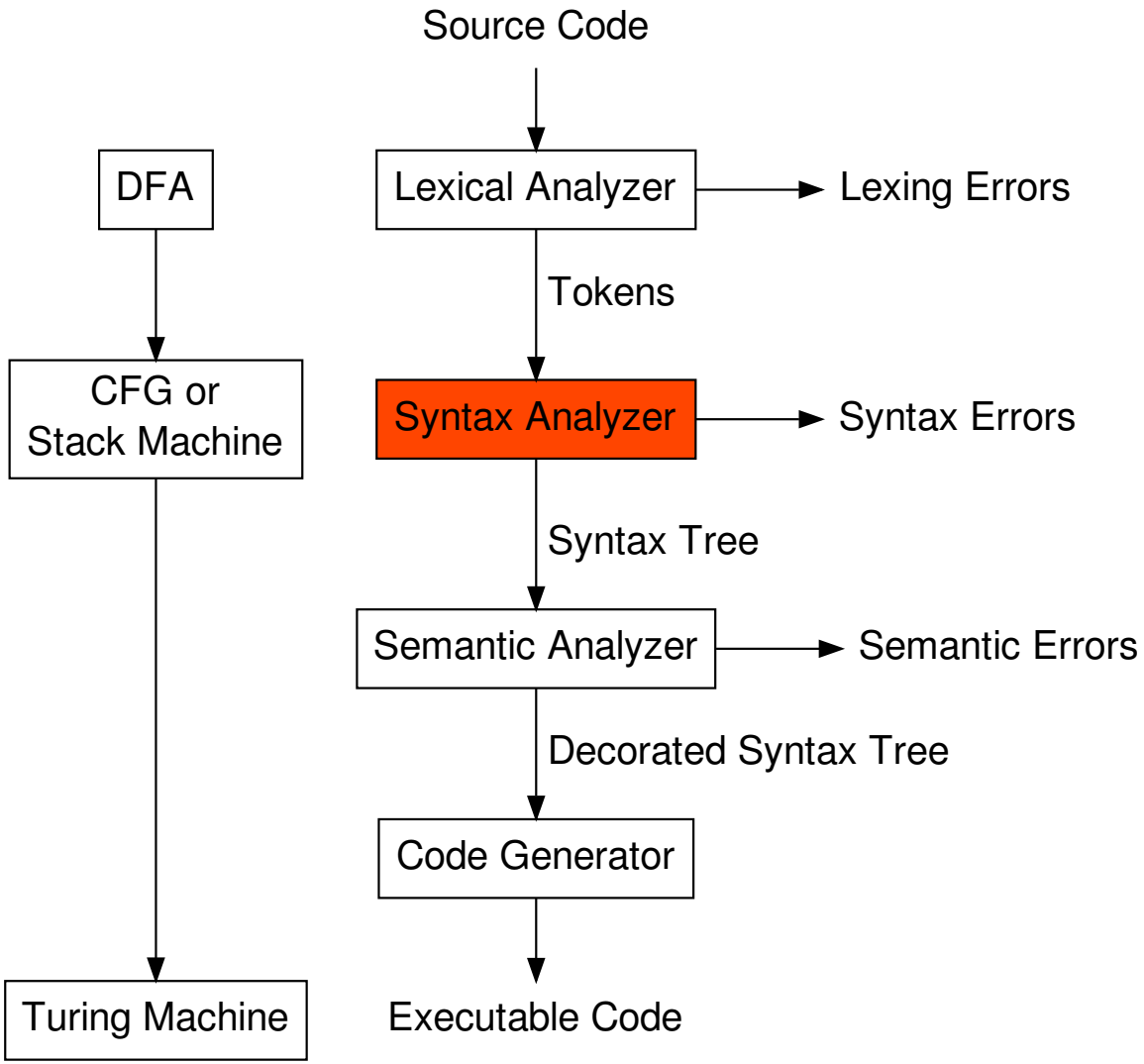
DFA Example

Input: `if iota`



Output: `IF, IDENT(iota),`

Syntax Analysis



Syntax Analysis

Problem: Convert a sequence of tokens to a **syntax tree**.

Syntax Analysis

Problem: Convert a sequence of tokens to a **syntax tree**.

- The tree encodes the grammatical structure of the program.

Syntax Analysis

Problem: Convert a sequence of tokens to a **syntax tree**.

- The tree encodes the grammatical structure of the program.
- We'll store further information about the program on the tree.

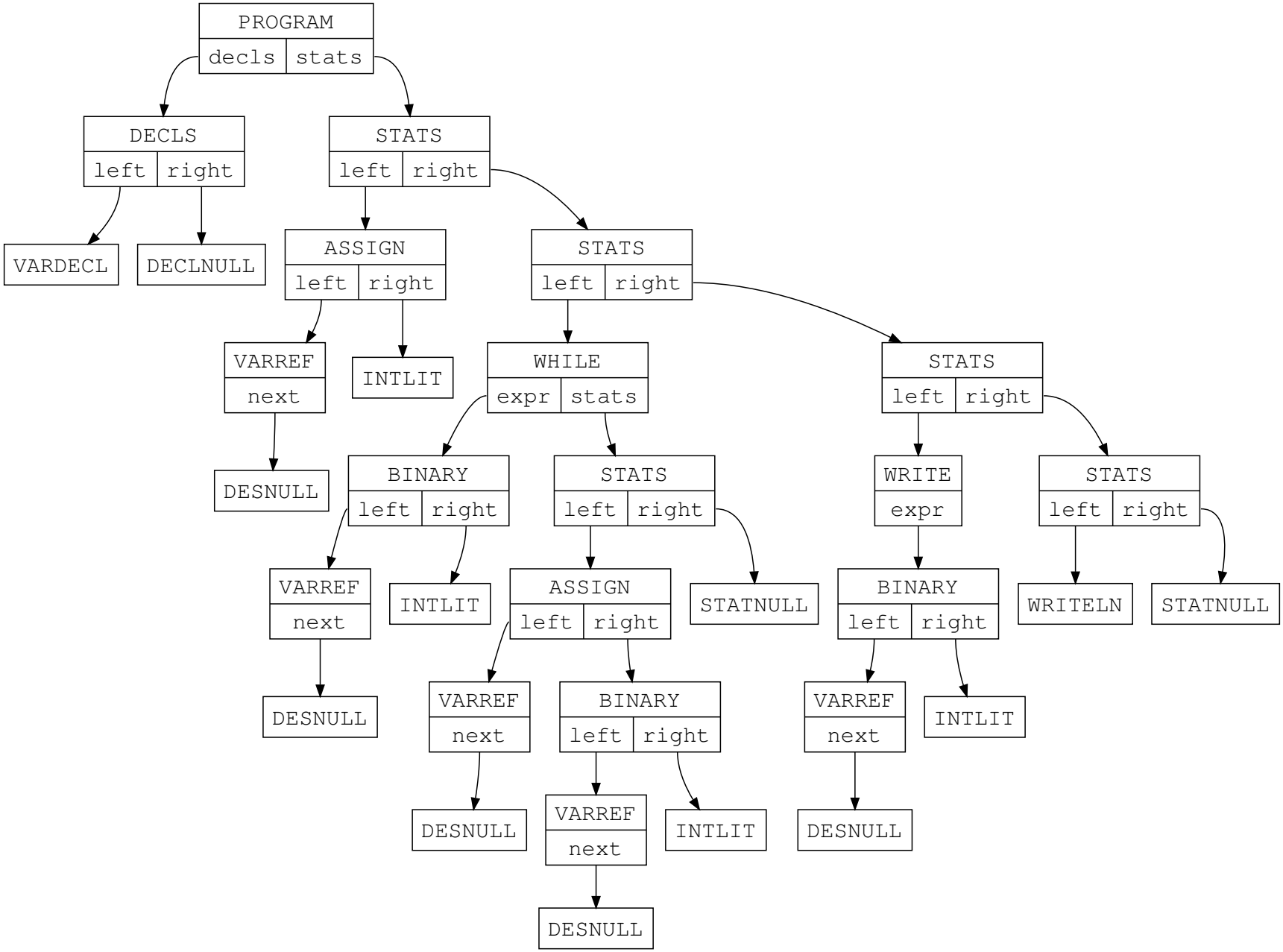
Syntax Analysis

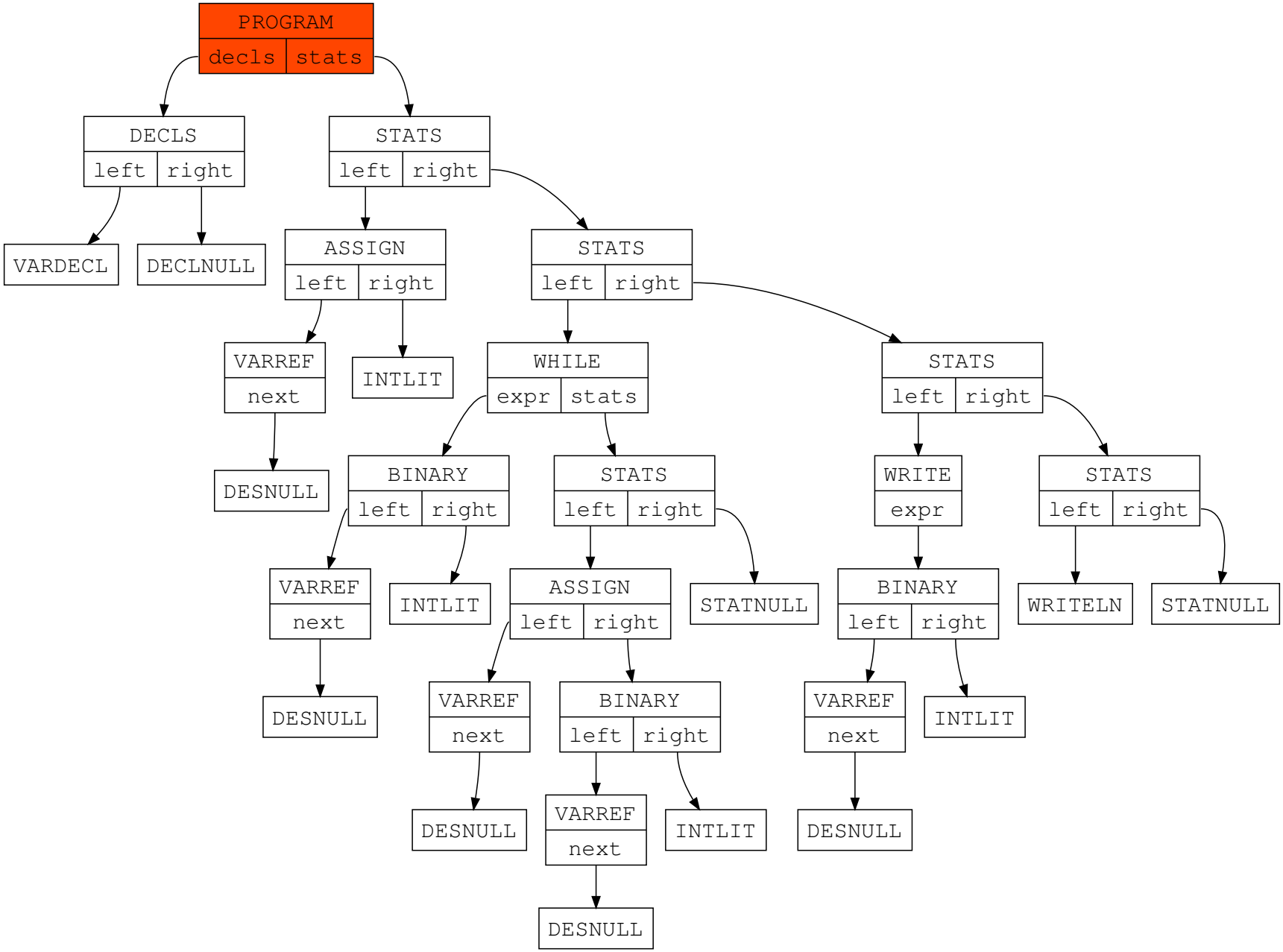
Problem: Convert a sequence of tokens to a **syntax tree**.

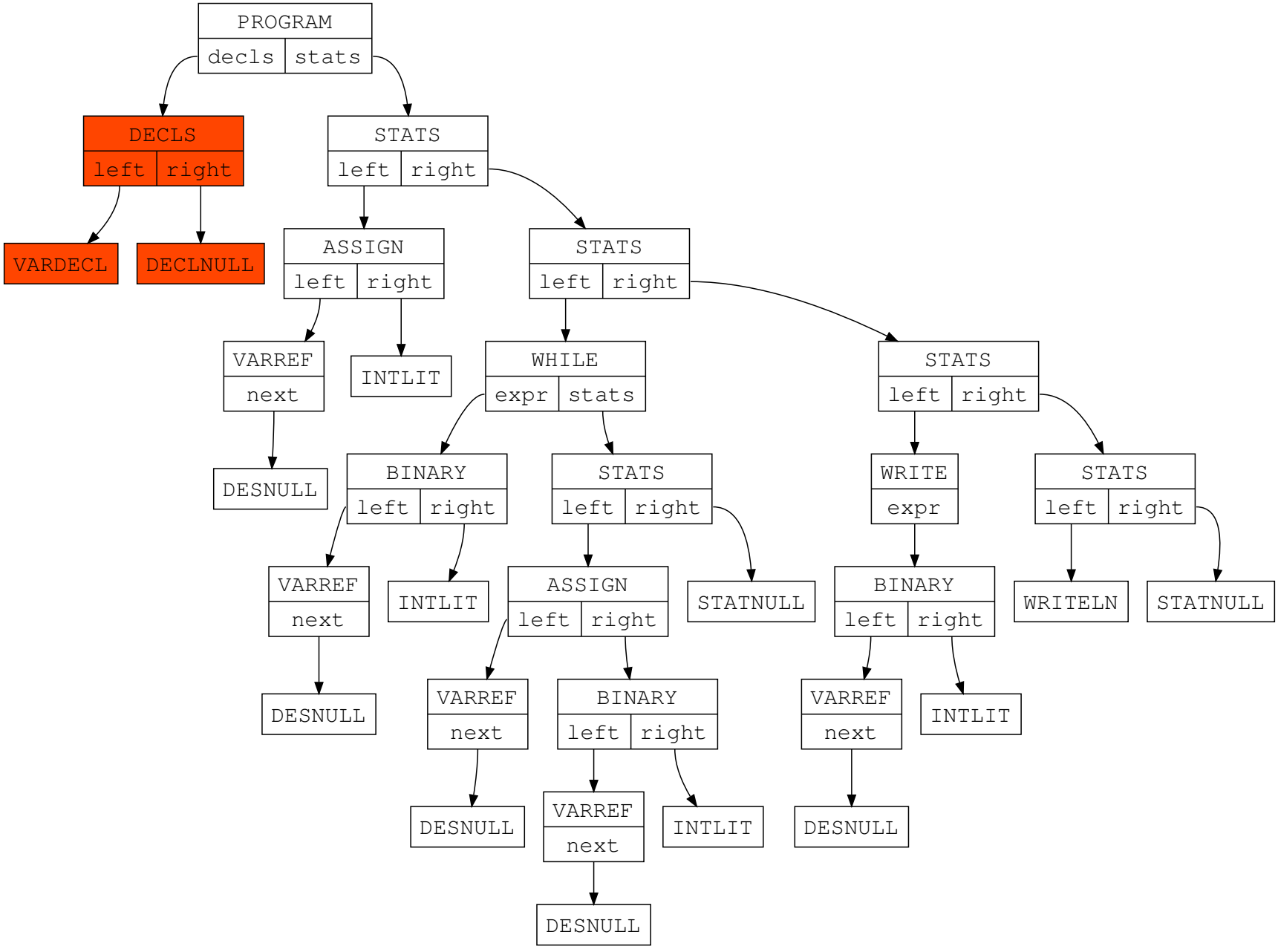
- The tree encodes the grammatical structure of the program.
- We'll store further information about the program on the tree.
- Syntax trees are **very** convenient for operating on programs.

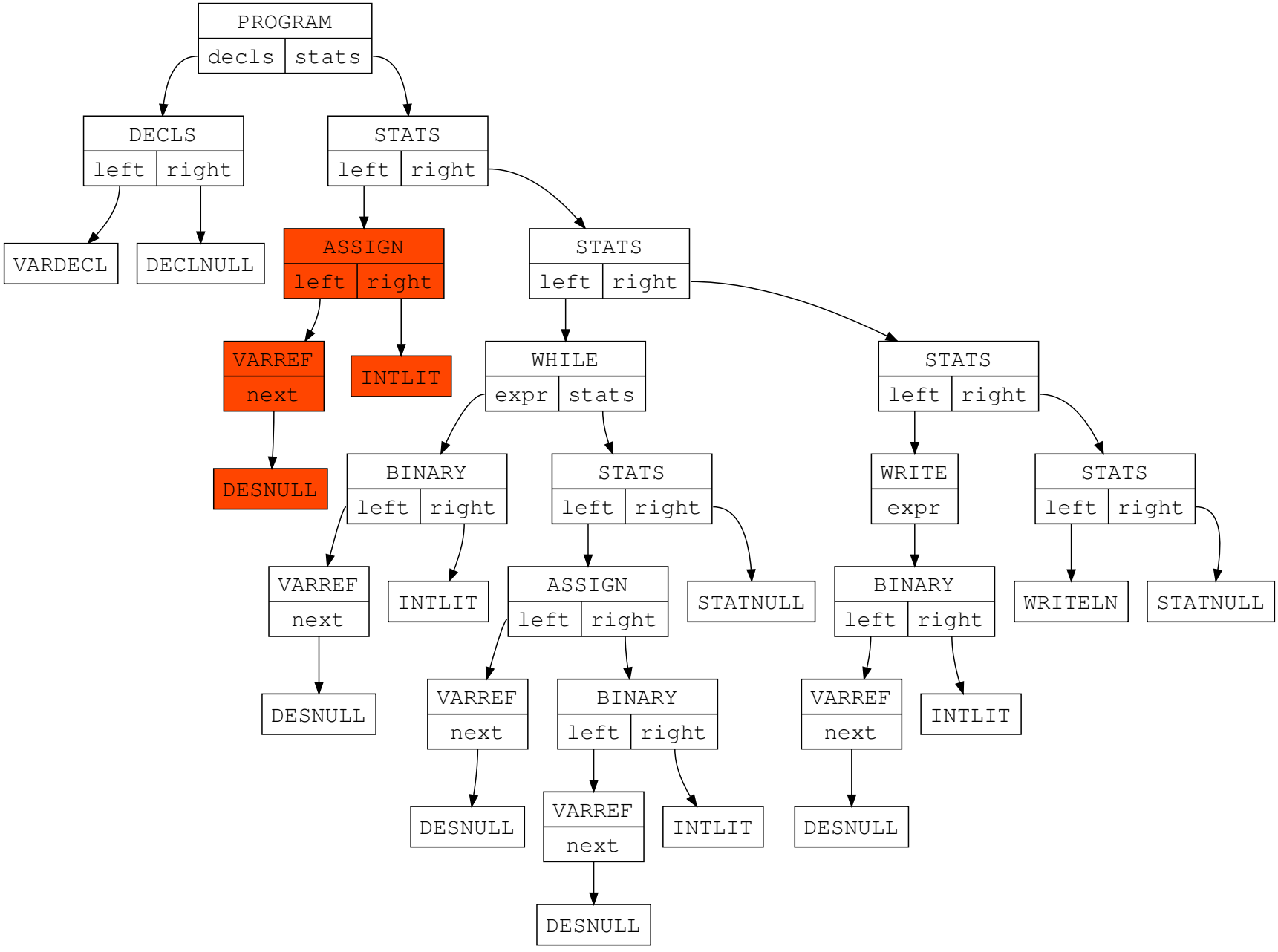
Example:

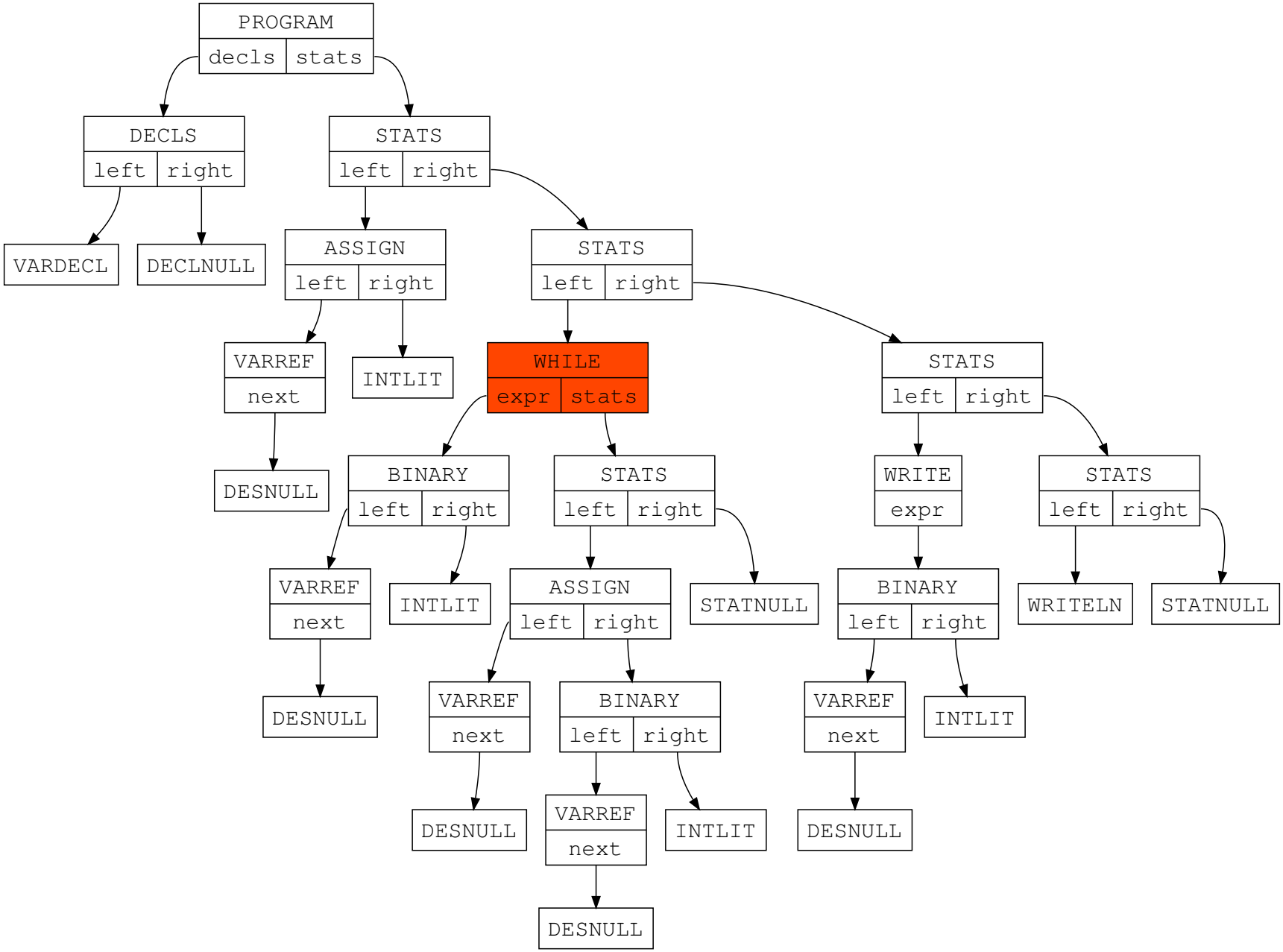
```
PROGRAM P;

VAR i : INTEGER;
BEGIN
    i := 1;
    WHILE i < 10 DO
        i := i + 1;
    ENDDO;
    WRITE i - 10; WRITELN;
END.
```

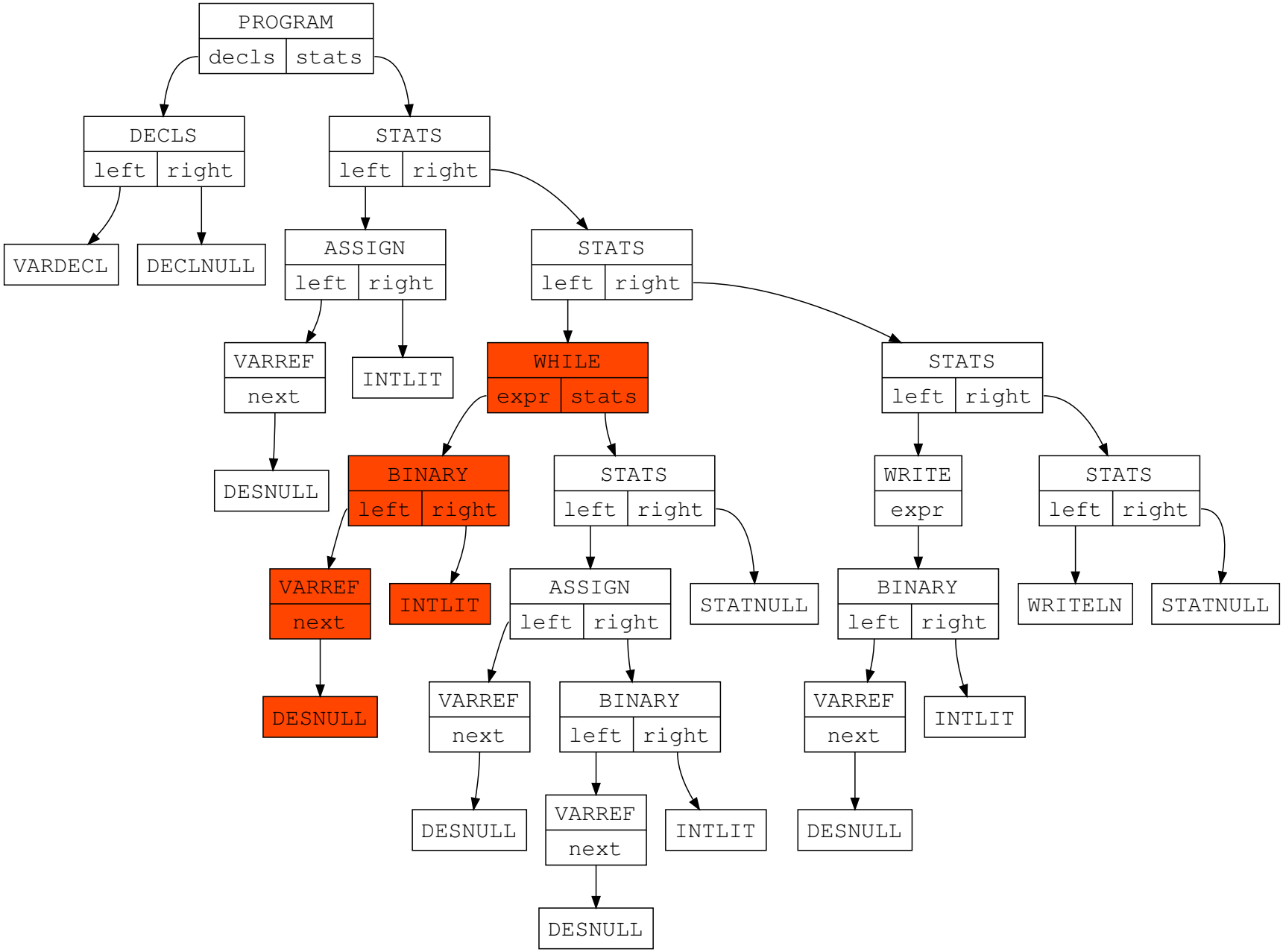

PROGRAM
decls
stats
DECLS
left
right
STATS
left
right
VARDECL
DECLNULL
ASSIGN
left
right
STATS
left
right
VARREF
next
INTLIT
WHILE
expr
stats
STATS
left
right
DESNULL
BINARY
left
right
STATS
left
right
WRITE
expr
STATS
left
right
VARREF
next
INTLIT
ASSIGN
left
right
STATNULL
BINARY
left
right
WRITELN
STATNULL
DESNULL
VARREF
next
BINARY
left
right
VARREF
next
INTLIT
DESNULL
VARREF
next
INTLIT
DESNULL
DESNULL

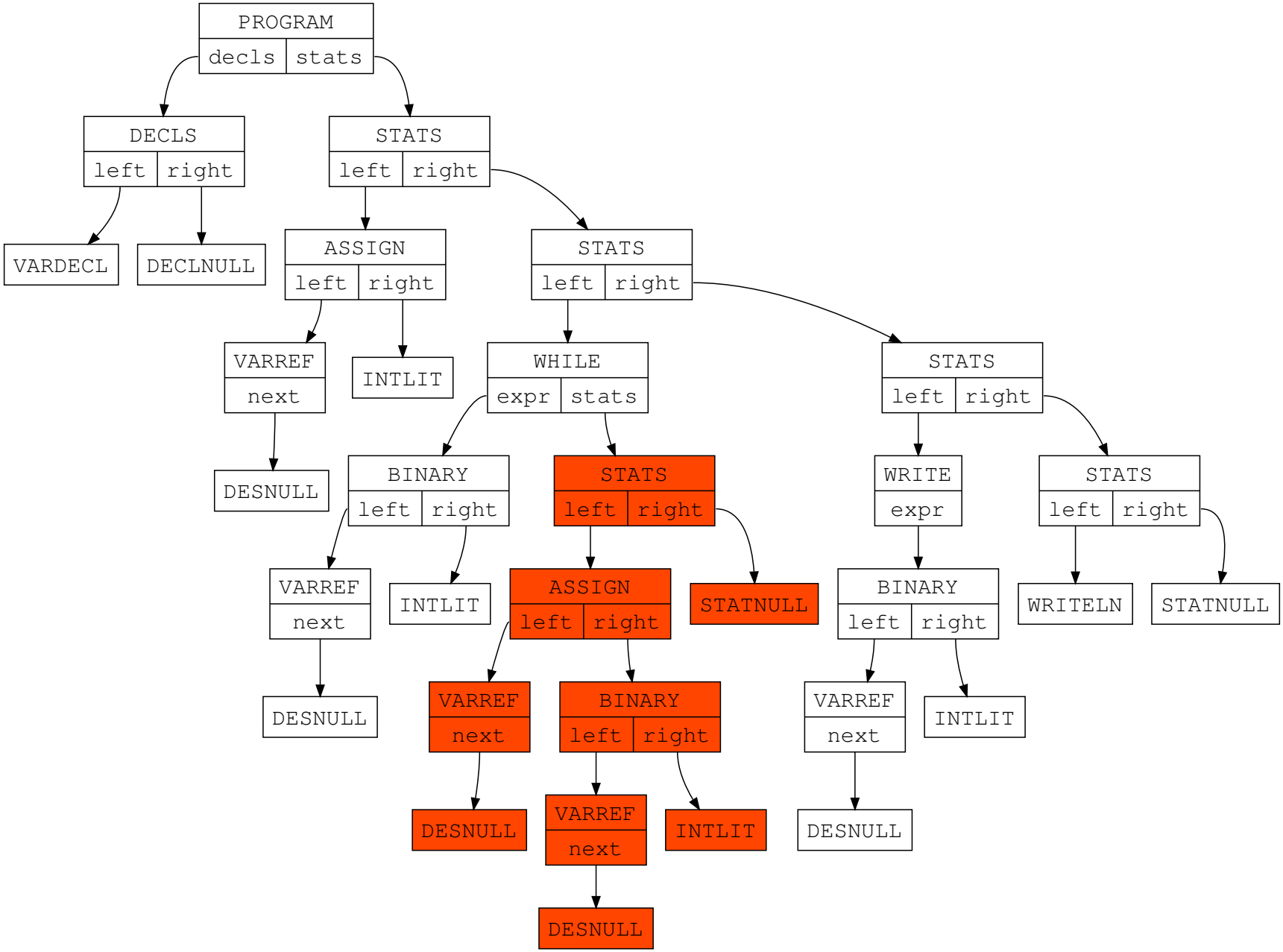

PROGRAM
decls
stats
DECLS
left
right
STATS
left
right
VARDECL
DECLNULL
ASSIGN
left
right
STATS
left
right
VARREF
next
INTLIT
WHILE
expr
stats
STATS
left
right
DESNULL
BINARY
left
right
STATS
left
right
WRITE
expr
STATS
left
right
VARREF
next
INTLIT
ASSIGN
left
right
STATNULL
BINARY
left
right
WRITELN
STATNULL
DESNULL
VARREF
next
BINARY
left
right
VARREF
next
INTLIT
DESNULL
VARREF
next
INTLIT
DESNULL
DESNULL

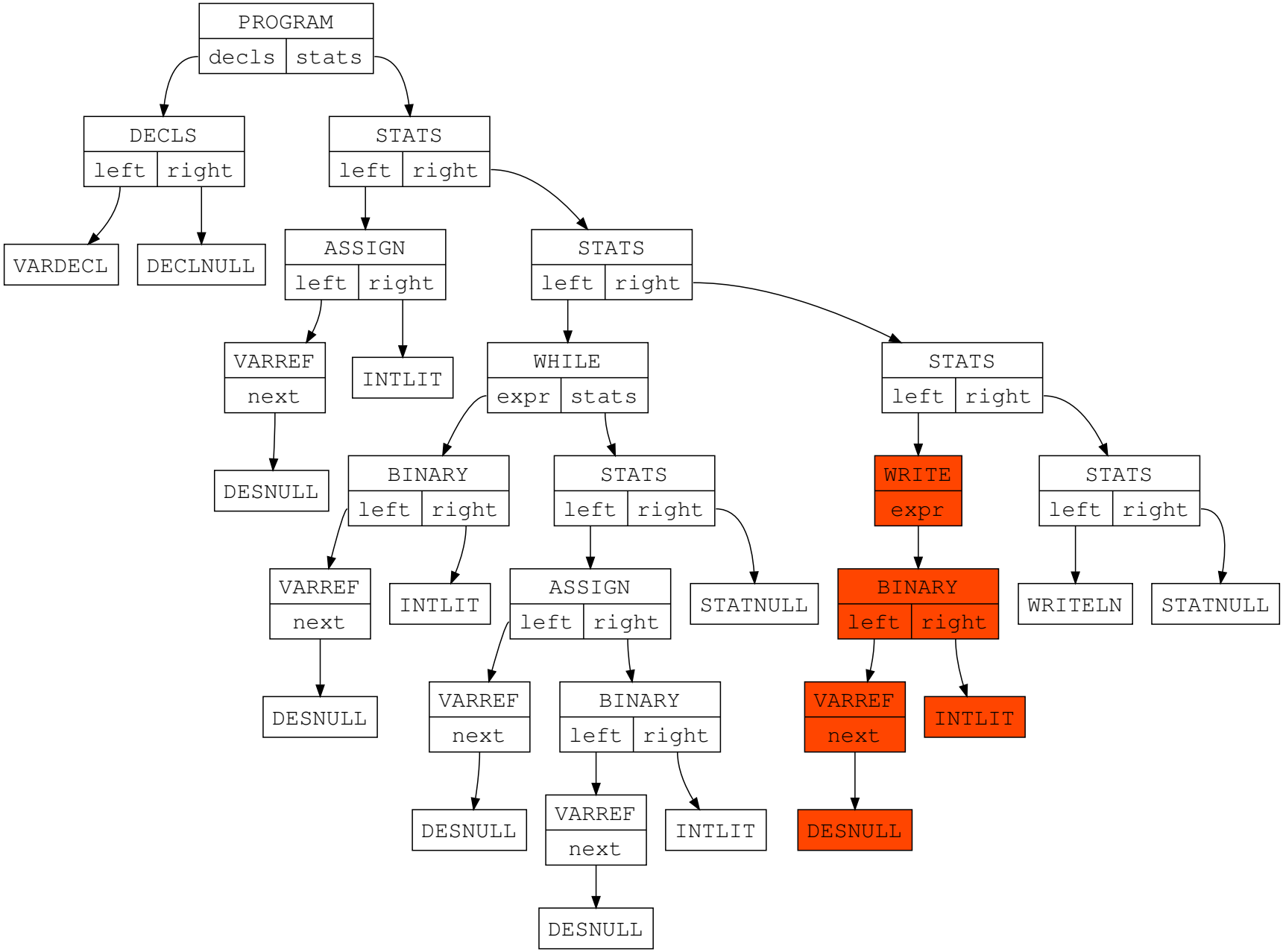

PROGRAM
decls
stats
DECLS
left
right
STATS
left
right
VARDECL
DECLNULL
ASSIGN
left
right
STATS
left
right
VARREF
next
INTLIT
WHILE
expr
stats
STATS
left
right
DESNULL
BINARY
left
right
STATS
left
right
WRITE
expr
STATS
left
right
VARREF
next
INTLIT
ASSIGN
left
right
STATNULL
BINARY
left
right
WRITELN
STATNULL
DESNULL
VARREF
next
BINARY
left
right
VARREF
next
INTLIT
DESNULL
VARREF
next
INTLIT
DESNULL
DESNULL

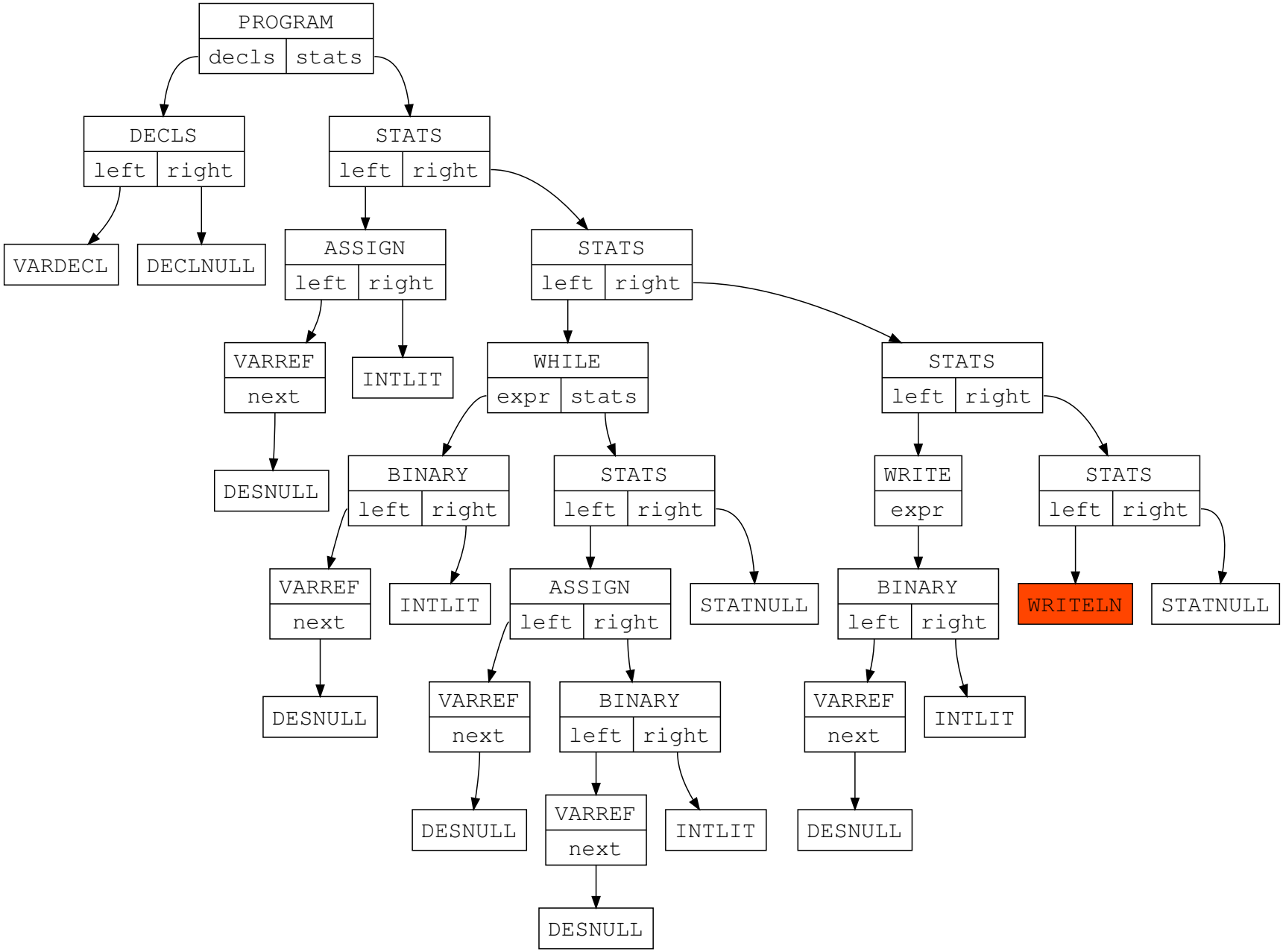

PROGRAM
decls
stats
DECLS
left
right
STATS
left
right
VARDECL
DECLNULL
ASSIGN
left
right
STATS
left
right
VARREF
next
INTLIT
WHILE
expr
stats
STATS
left
right
DESNULL
BINARY
left
right
STATS
left
right
WRITE
expr
STATS
left
right
VARREF
next
INTLIT
ASSIGN
left
right
STATNULL
BINARY
left
right
WRITELN
STATNULL
DESNULL
VARREF
next
BINARY
left
right
VARREF
next
INTLIT
DESNULL
VARREF
next
BINARY
left
right
DESNULL
VARREF
next
INTLIT
DESNULL


PROGRAM
decls
stats
DECLS
left
right
STATS
left
right
VARDECL
DECLNULL
ASSIGN
left
right
STATS
left
right
VARREF
next
INTLIT
WHILE
expr
stats
STATS
left
right
DESNULL
BINARY
left
right
STATS
left
right
WRITE
expr
STATS
left
right
VARREF
next
INTLIT
ASSIGN
left
right
STATNULL
BINARY
left
right
WRITELN
STATNULL
DESNULL
VARREF
next
BINARY
left
right
VARREF
next
INTLIT
DESNULL
VARREF
next
INTLIT
DESNULL
DESNULL

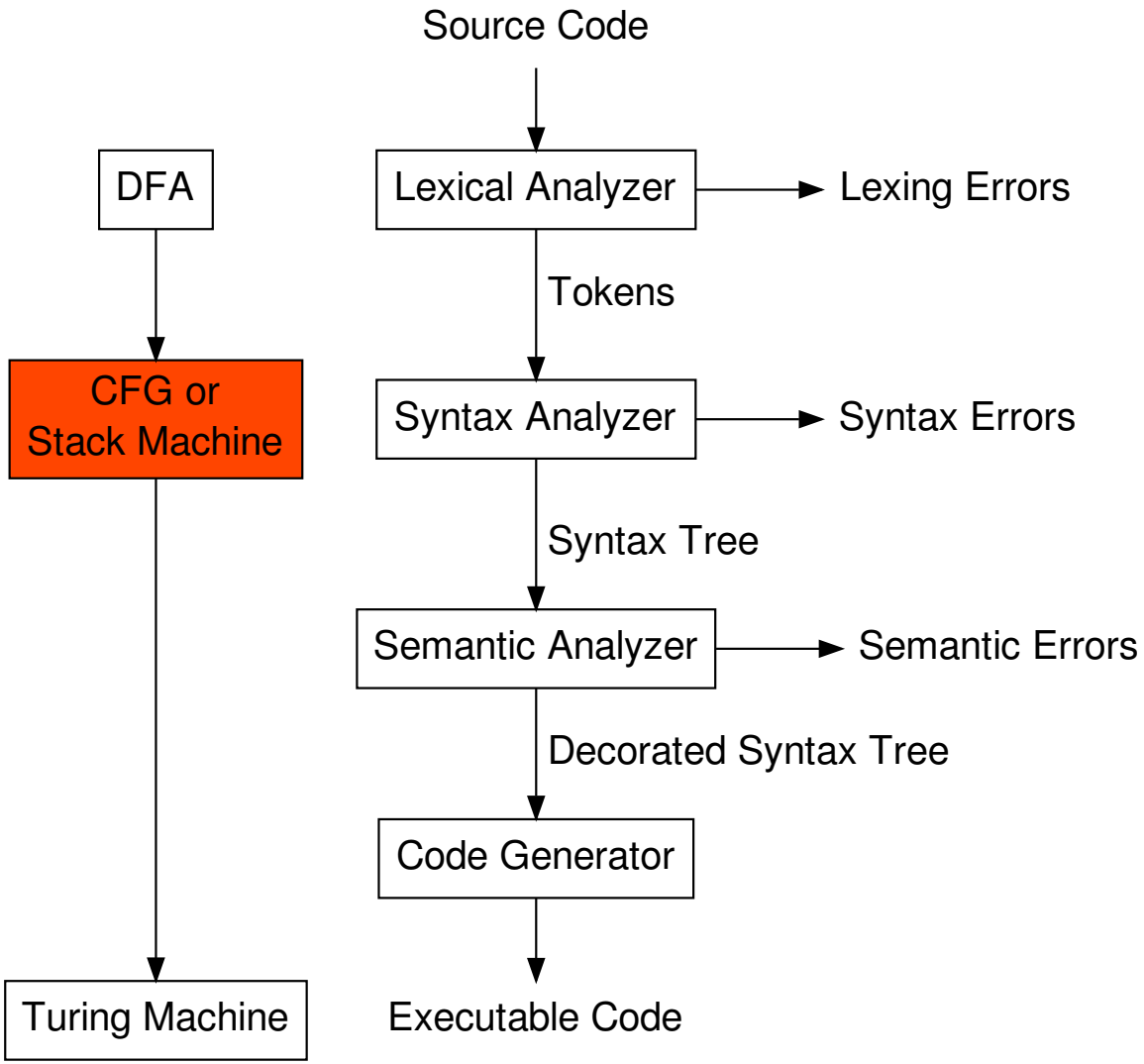

PROGRAM
decls
stats
DECLS
left
right
STATS
left
right
VARDECL
DECLNULL
ASSIGN
left
right
STATS
left
right
VARREF
next
INTLIT
WHILE
expr
stats
STATS
left
right
DESNULL
BINARY
left
right
STATS
left
right
WRITE
expr
STATS
left
right
VARREF
next
INTLIT
ASSIGN
left
right
STATNULL
BINARY
left
right
WRITELN
STATNULL
DESNULL
VARREF
next
BINARY
left
right
VARREF
next
INTLIT
DESNULL
VARREF
next
BINARY
left
right
DESNULL
VARREF
next
INTLIT
DESNULL


PROGRAM
decls
stats
DECLS
left
right
STATS
left
right
VARDECL
DECLNULL
ASSIGN
left
right
STATS
left
right
VARREF
next
INTLIT
WHILE
expr
stats
STATS
left
right
DESNULL
BINARY
left
right
STATS
left
right
WRITE
expr
STATS
left
right
VARREF
next
INTLIT
ASSIGN
left
right
STATNULL
BINARY
left
right
WRITELN
STATNULL
DESNULL
VARREF
next
BINARY
left
right
VARREF
next
INTLIT
DESNULL
VARREF
next
INTLIT
DESNULL
DESNULL


PROGRAM
decls
stats
DECLS
left
right
STATS
left
right
VARDECL
DECLNULL
ASSIGN
left
right
STATS
left
right
VARREF
next
INTLIT
WHILE
expr
stats
STATS
left
right
DESNULL
BINARY
left
right
STATS
left
right
WRITE
expr
STATS
left
right
VARREF
next
INTLIT
ASSIGN
left
right
STATNULL
BINARY
left
right
WRITELN
STATNULL
DESNULL
VARREF
next
BINARY
left
right
VARREF
next
INTLIT
DESNULL
VARREF
next
INTLIT
DESNULL
VARREF
next
DESNULL


PROGRAM
decls
stats
DECLS
left
right
STATS
left
right
VARDECL
DECLNULL
ASSIGN
left
right
STATS
left
right
VARREF
next
INTLIT
WHILE
expr
stats
STATS
left
right
DESNULL
BINARY
left
right
STATS
left
right
WRITE
expr
STATS
left
right
VARREF
next
INTLIT
ASSIGN
left
right
STATNULL
BINARY
left
right
WRITELN
STATNULL
DESNULL
VARREF
next
BINARY
left
right
VARREF
next
INTLIT
DESNULL
VARREF
next
INTLIT
DESNULL
DESNULL

Stack Machines / Context Free Grammars



Stack Machines / Context Free Grammars

Stack Machines / Context Free Grammars

- We specify syntax with a **Context Free Grammar** (CFG).

Stack Machines / Context Free Grammars

- We specify syntax with a **Context Free Grammar** (CFG).
- Programs have a single grammatical structure, regardless of context.

Stack Machines / Context Free Grammars

- We specify syntax with a **Context Free Grammar** (CFG).
- Programs have a single grammatical structure, regardless of context.
- English is not context free: “Fruit flies like a banana.”

Stack Machines / Context Free Grammars

- We specify syntax with a **Context Free Grammar** (CFG).
- Programs have a single grammatical structure, regardless of context.
- English is not context free: “Fruit flies like a banana.”

Stack Machines / Context Free Grammars

- We specify syntax with a **Context Free Grammar** (CFG).
- Programs have a single grammatical structure, regardless of context.
- English is not context free: "Fruit flies like a banana."
  - Context A: "Fruit" is the subject, "flies" is the verb.

Stack Machines / Context Free Grammars

- We specify syntax with a **Context Free Grammar** (CFG).
- Programs have a single grammatical structure, regardless of context.
- English is not context free: "Fruit flies like a banana."
  - Context A: "Fruit" is the subject, "flies" is the verb.
  - Context B: "Fruit flies" is the subject, "like" is the verb.

Stack Machines / Context Free Grammars

- We specify syntax with a **Context Free Grammar** (CFG).
- Programs have a single grammatical structure, regardless of context.
- English is not context free: "Fruit flies like a banana."
  - Context A: "Fruit" is the subject, "flies" is the verb.
  - Context B: "Fruit flies" is the subject, "like" is the verb.

Stack Machines / Context Free Grammars

- We specify syntax with a **Context Free Grammar** (CFG).
- Programs have a single grammatical structure, regardless of context.
- English is not context free: "Fruit flies like a banana."
  - Context A: "Fruit" is the subject, "flies" is the verb.
  - Context B: "Fruit flies" is the subject, "like" is the verb.

Example

Tokens:

$$\Sigma = \{\texttt{while}, \texttt{do}, \texttt{done}, \texttt{true}, \texttt{or}, \texttt{not}, \texttt{:=}, \texttt{;}, \texttt{(}, \texttt{)}, \texttt{x}, \texttt{y}, \texttt{z}\}$$

Grammar:

$$\begin{aligned}
\text{Stats} &\rightarrow \text{Stat}\ \ \texttt{;}\ \ \text{Stats} \mid \varepsilon \\
\text{Stat} &\rightarrow \text{Ident}\ \ \texttt{:=}\ \ \text{Exp} \mid \texttt{while}\ \ \text{Exp}\ \ \texttt{do}\ \ \text{Stats}\ \ \texttt{done} \\
\text{Exp} &\rightarrow \texttt{(}\text{Exp}\ \ \texttt{or}\ \ \text{Exp}\texttt{)} \mid \texttt{(not}\ \ \text{Exp}\texttt{)} \mid \text{Ident} \mid \texttt{true} \\
\text{Ident} &\rightarrow \ \ \texttt{x} \mid \texttt{y} \mid \texttt{z}
\end{aligned}$$

Example Program:

```
while (not x) do x := (not x); done; y := x;
```

Stack Machines

A recipe for creating automatons:

Stack Machines

A recipe for creating automatons:

- Take a finite directed graph G.

Stack Machines

A recipe for creating automatons:

- Take a finite directed graph G.
- Add a data structure.

Stack Machines

A recipe for creating automatons:

- Take a finite directed graph G.
- Add a data structure.
- Decorate edges of G with symbols of the input alphabet and operations on the data structure.

Stack Machines

A recipe for creating automatons:

- Take a finite directed graph G.
- Add a data structure.
- Decorate edges of G with symbols of the input alphabet and operations on the data structure.

Stack Machines

A recipe for creating automatons:

- Take a finite directed graph G.
- Add a data structure.
- Decorate edges of G with symbols of the input alphabet and operations on the data structure.

For **Stack Machines** the added data structure is a stack. We may:

Stack Machines

A recipe for creating automatons:

- Take a finite directed graph G.
- Add a data structure.
- Decorate edges of G with symbols of the input alphabet and operations on the data structure.

For **Stack Machines** the added data structure is a stack. We may:

- Push symbols onto the stack.

Stack Machines

A recipe for creating automatons:

- Take a finite directed graph G.
- Add a data structure.
- Decorate edges of G with symbols of the input alphabet and operations on the data structure.

For **Stack Machines** the added data structure is a stack. We may:

- Push symbols onto the stack.
- Pop symbols off the stack.

Stack Machines

A recipe for creating automatons:

- Take a finite directed graph G.
- Add a data structure.
- Decorate edges of G with symbols of the input alphabet and operations on the data structure.

For **Stack Machines** the added data structure is a stack. We may:

- Push symbols onto the stack.
- Pop symbols off the stack.
- Peek at the top stack symbol.

Stack Machines

A recipe for creating automatons:

- Take a finite directed graph G.
- Add a data structure.
- Decorate edges of G with symbols of the input alphabet and operations on the data structure.

For **Stack Machines** the added data structure is a stack. We may:

- Push symbols onto the stack.
- Pop symbols off the stack.
- Peek at the top stack symbol.

Stack Machines

A recipe for creating automatons:

- Take a finite directed graph G.
- Add a data structure.
- Decorate edges of G with symbols of the input alphabet and operations on the data structure.

For **Stack Machines** the added data structure is a stack. We may:

- Push symbols onto the stack.
- Pop symbols off the stack.
- Peek at the top stack symbol.

Notation:

Stack Machines

A recipe for creating automatons:

- Take a finite directed graph G.
- Add a data structure.
- Decorate edges of G with symbols of the input alphabet and operations on the data structure.

For **Stack Machines** the added data structure is a stack. We may:

- Push symbols onto the stack.
- Pop symbols off the stack.
- Peek at the top stack symbol.

Notation:

- \$: The end of the input string.

Stack Machines

A recipe for creating automatons:

- Take a finite directed graph G.
- Add a data structure.
- Decorate edges of G with symbols of the input alphabet and operations on the data structure.

For **Stack Machines** the added data structure is a stack. We may:

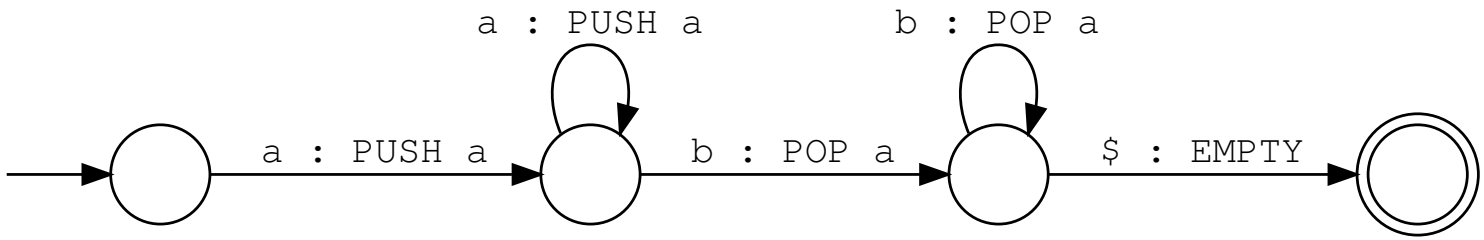
- Push symbols onto the stack.
- Pop symbols off the stack.
- Peek at the top stack symbol.

Notation:

- \$: The end of the input string.
- Edges decorated with "x : [PUSH/POP/PEEK] y", where x and y are tokens.

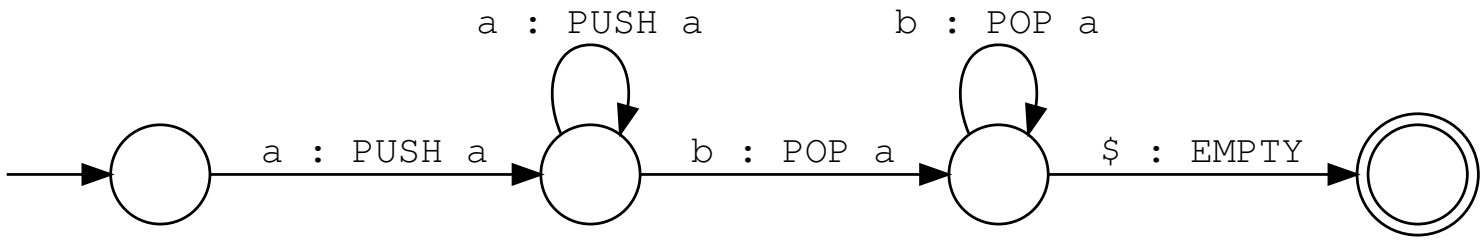
Example

A stack machine to recognize $\{a^n b^n : n \in \mathbb{N}\}$:



Example

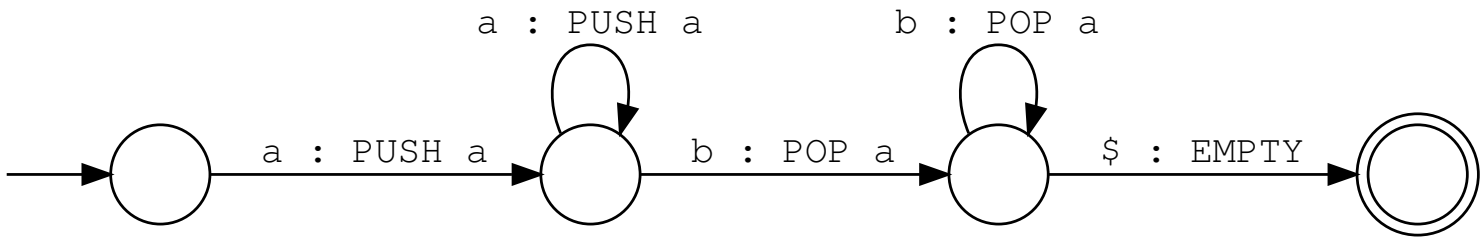
A stack machine to recognize $\{a^n b^n : n \in \mathbb{N}\}$:



Here, the stack lets us:

Example

A stack machine to recognize $\{a^n b^n : n \in \mathbb{N}\}$:

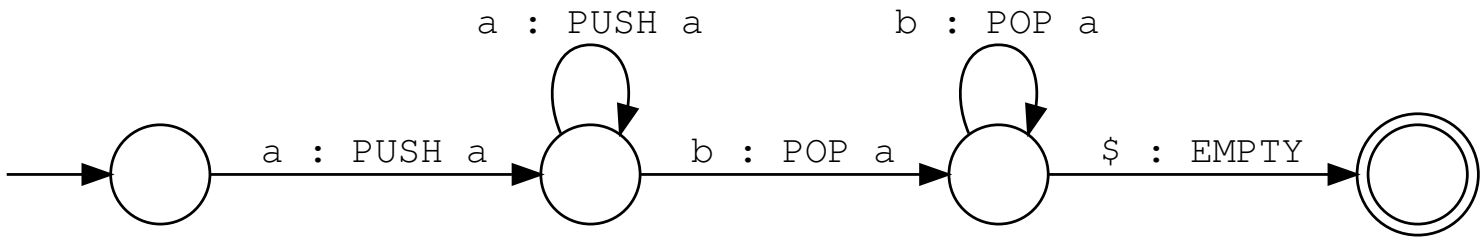


Here, the stack lets us:

- Count up the number of a's.

Example

A stack machine to recognize $\{a^n b^n : n \in \mathbb{N}\}$:

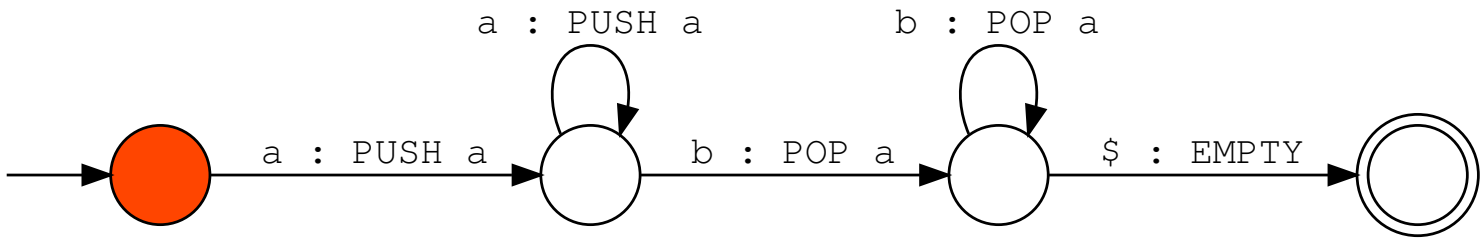


Here, the stack lets us:

- Count up the number of a's.
- Count off the number of b's.

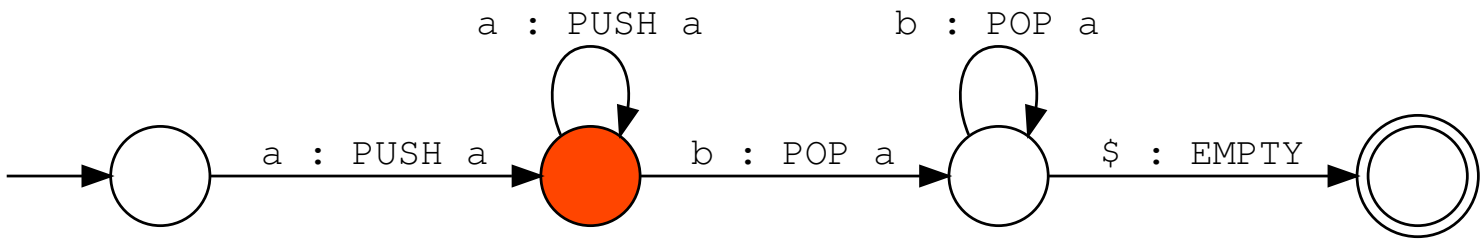
Example

Input: aabb$
Stack:



Example

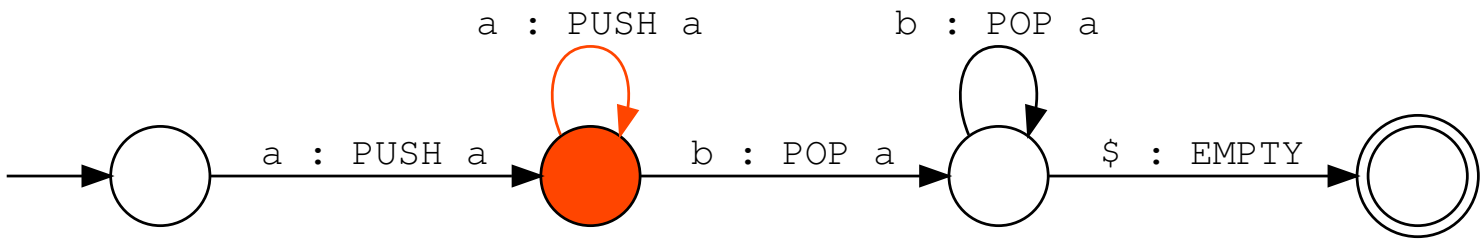
Input: aabb$
Stack: a



Example

Input: aabb$

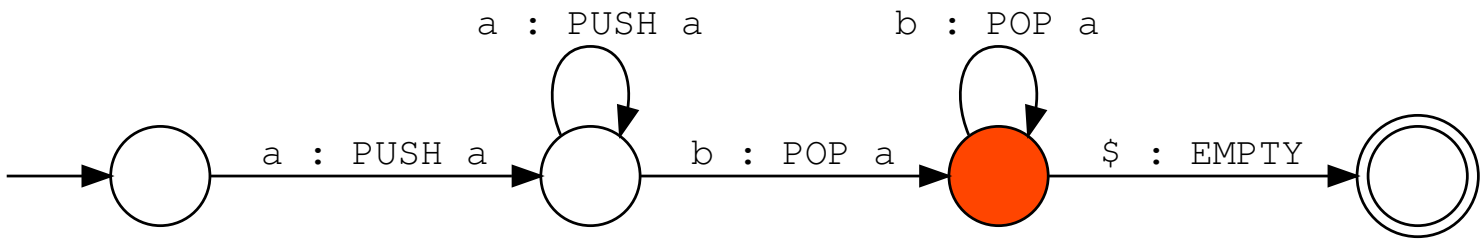
Stack: aa



Example

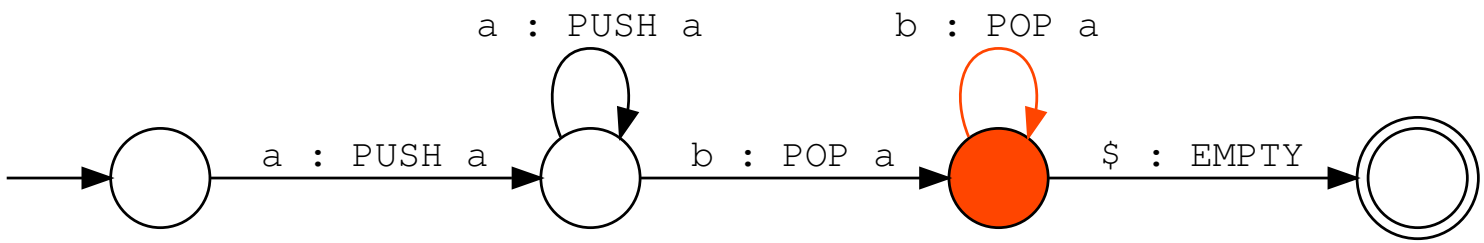
Input: aabb$

Stack: a



Example

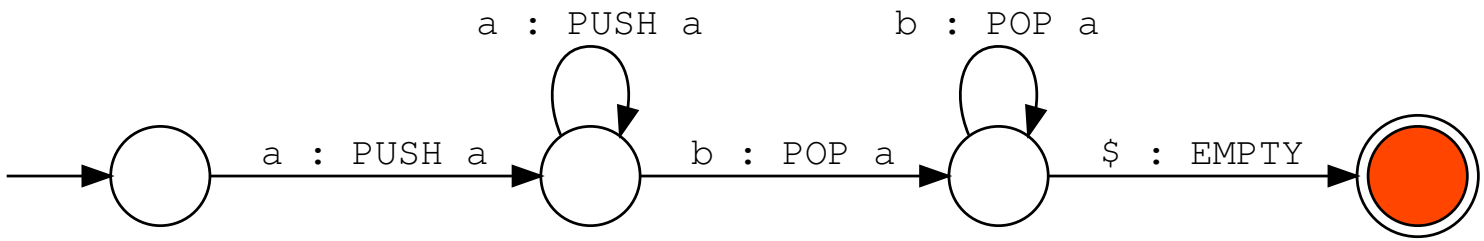
Input: aabb$
Stack:



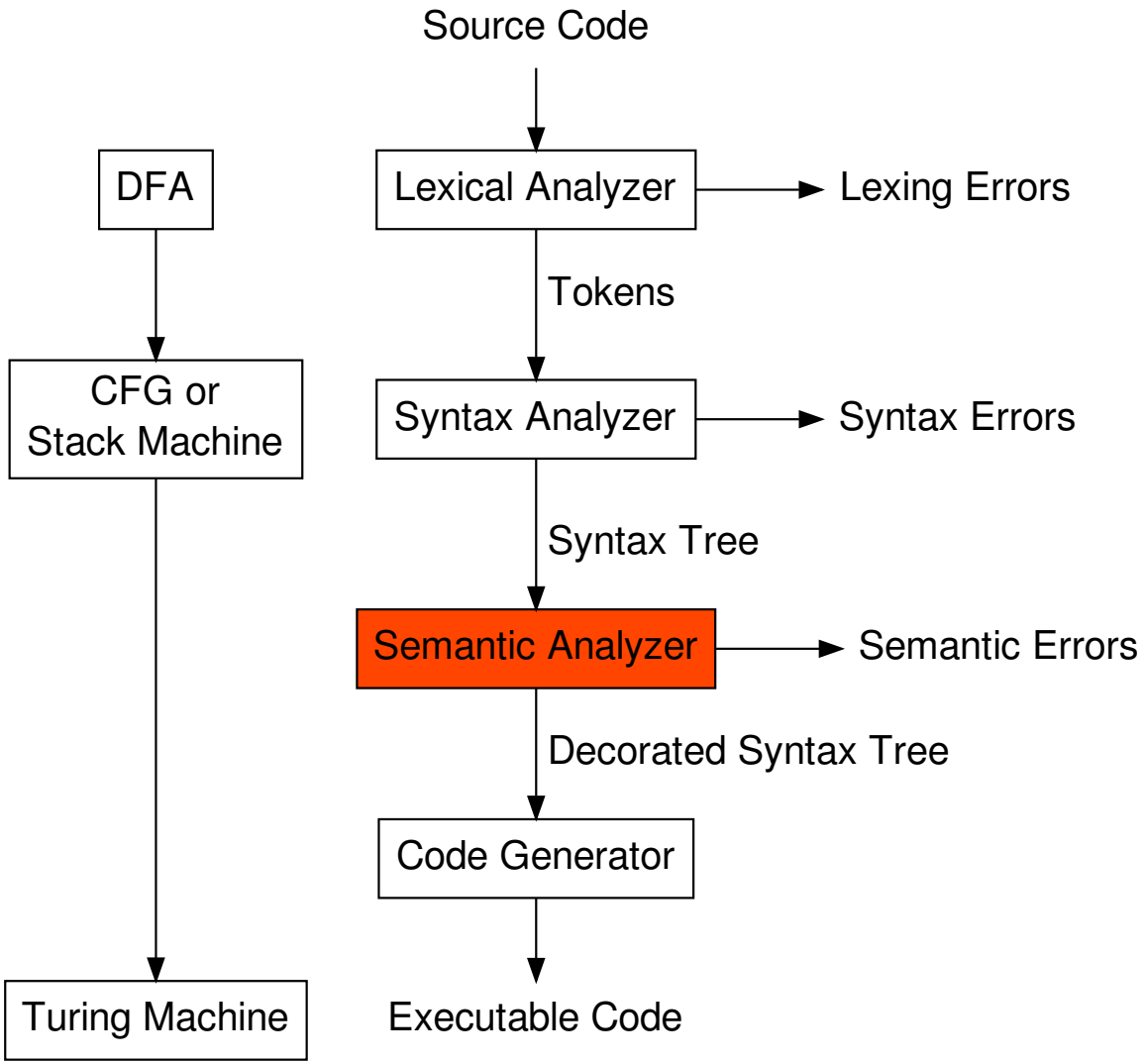
Example

Input: aabb$

Stack:



Semantic Analysis



Semantic Analysis

We want compilers that...

Semantic Analysis

We want compilers that...

- ...help us avoid making silly mistakes.

Semantic Analysis

We want compilers that...

- ...help us avoid making silly mistakes.
- ...produce the program we expect.

Semantic Analysis

We want compilers that...

- ...help us avoid making silly mistakes.
- ...produce the program we expect.

Semantic Analysis

We want compilers that...

- ...help us avoid making silly mistakes.
- ...produce the program we expect.

During the semantic analysis step we gather information about:

Semantic Analysis

We want compilers that...

- ...help us avoid making silly mistakes.
- ...produce the program we expect.

During the semantic analysis step we gather information about:

- Variables

Semantic Analysis

We want compilers that...

- ...help us avoid making silly mistakes.
- ...produce the program we expect.

During the semantic analysis step we gather information about:

- Variables
- Scope

Semantic Analysis

We want compilers that...

- ...help us avoid making silly mistakes.
- ...produce the program we expect.

During the semantic analysis step we gather information about:

- Variables
- Scope
- Type — what a variable is supposed to represent.

Semantic Analysis

This is one of the more "language dependent" parts of a compiler. Assuming a strongly typed language (like C), we will:

Semantic Analysis

This is one of the more "language dependent" parts of a compiler. Assuming a strongly typed language (like C), we will:

- Build data structures that categorize variables and procedure calls.

Semantic Analysis

This is one of the more "language dependent" parts of a compiler. Assuming a strongly typed language (like C), we will:

- Build data structures that categorize variables and procedure calls.
- Traverse the syntax tree:

Semantic Analysis

This is one of the more "language dependent" parts of a compiler. Assuming a strongly typed language (like C), we will:

- Build data structures that categorize variables and procedure calls.
- Traverse the syntax tree:

Semantic Analysis

This is one of the more "language dependent" parts of a compiler. Assuming a strongly typed language (like C), we will:

- Build data structures that categorize variables and procedure calls.
- Traverse the syntax tree:
  - Accumulate data from declarations.

Semantic Analysis

This is one of the more "language dependent" parts of a compiler. Assuming a strongly typed language (like C), we will:

- Build data structures that categorize variables and procedure calls.
- Traverse the syntax tree:
  - Accumulate data from declarations.
  - Examine statements, making sure they're well formed.

Semantic Analysis

Semantic Analysis

Examples of errors:

Semantic Analysis

Examples of errors:

- Using a variable/procedure without declaring it.

Semantic Analysis

Examples of errors:

- Using a variable/procedure without declaring it.
- Using a variable in a way that doesn't agree with it's type (Ex. adding a **string** to an **int**).

Semantic Analysis

Examples of errors:

- Using a variable/procedure without declaring it.
- Using a variable in a way that doesn't agree with it's type (Ex. adding a **string** to an **int**).
- Declaring a variable several times in the same scope, with different types.

Semantic Analysis

Examples of errors:

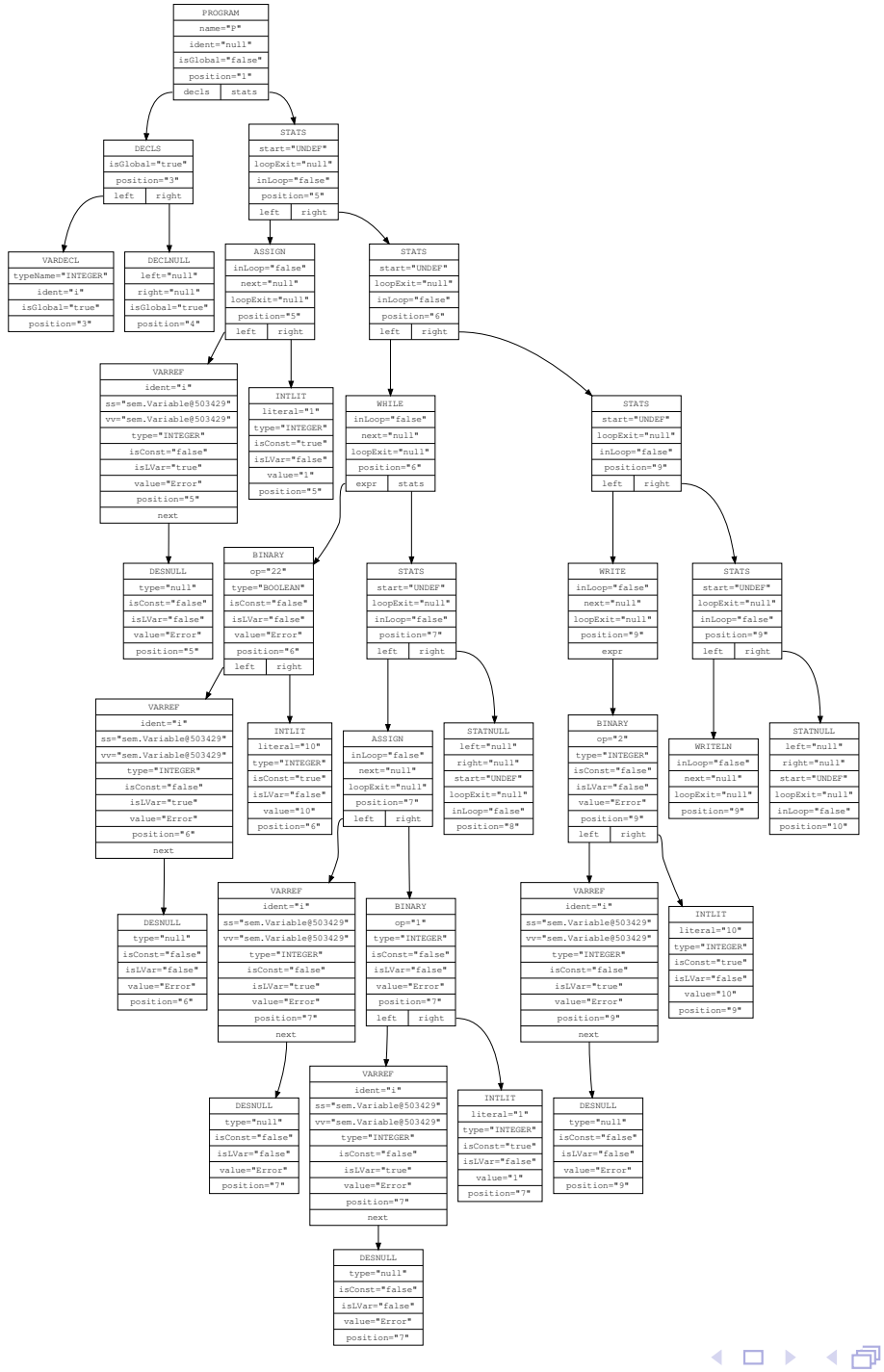
- Using a variable/procedure without declaring it.
- Using a variable in a way that doesn't agree with it's type (Ex. adding a **string** to an **int**).
- Declaring a variable several times in the same scope, with different types.

Semantic Analysis

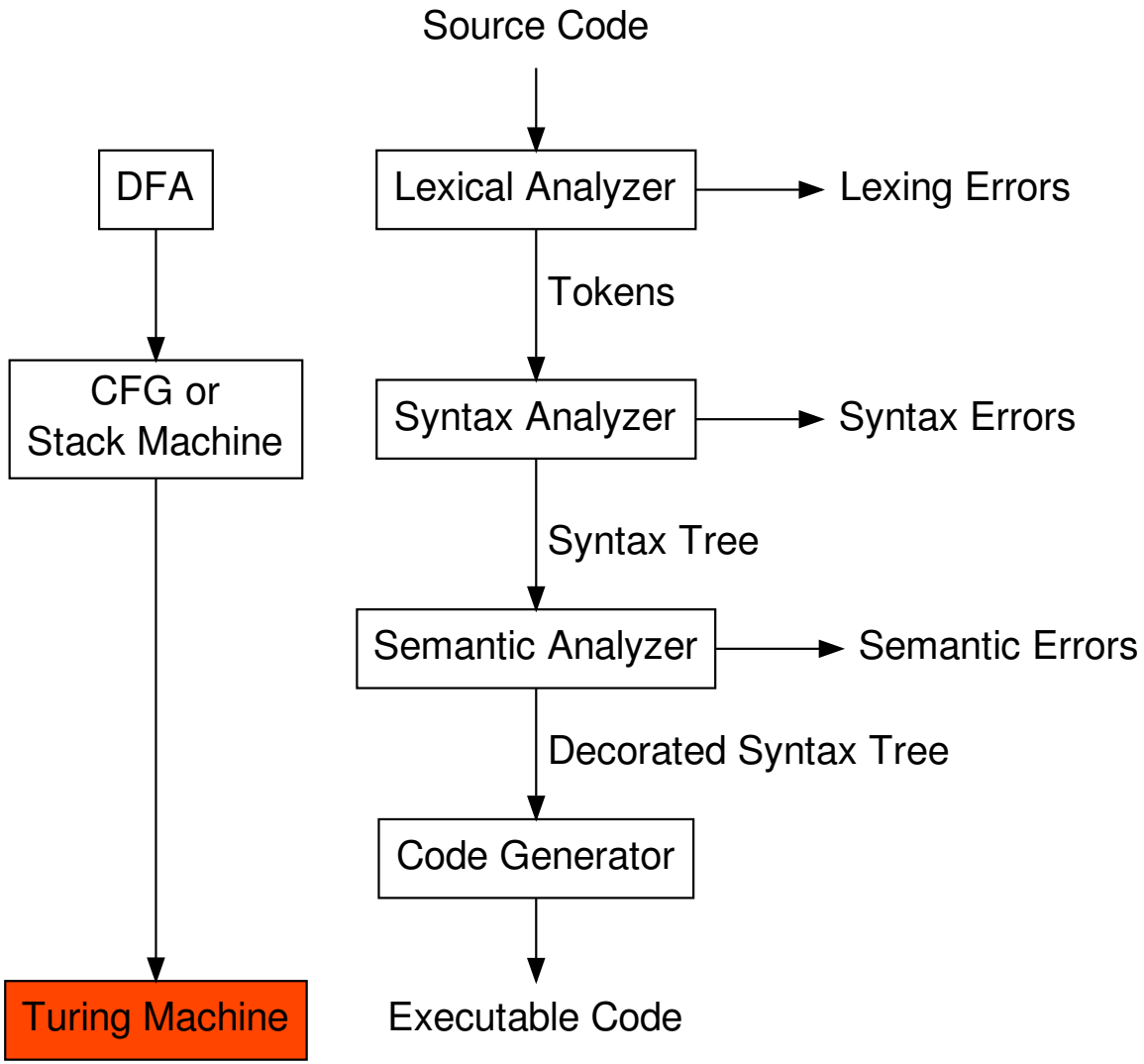
Examples of errors:

- Using a variable/procedure without declaring it.
- Using a variable in a way that doesn't agree with it's type (Ex. adding a **string** to an **int**).
- Declaring a variable several times in the same scope, with different types.

Important: The more you know at compile-time, the more efficient you can make your machine/byte code.


PROGRAM
name="p"
ident="null"
isGlobal="false"
position="1"
decls
stats
DECLS
isGlobal="true"
position="3"
left
right
STATS
start="UNDEF"
loopExit="null"
inLoop="false"
position="5"
left
right
VARDECL
typeName="INTEGER"
ident="i"
isGlobal="true"
position="3"
DECLNULL
left="null"
right="null"
isGlobal="true"
position="4"
ASSIGN
inLoop="false"
next="null"
loopExit="null"
position="5"
left
right
STATS
start="UNDEF"
loopExit="null"
inLoop="false"
position="6"
left
right
VARREF
ident="i"
ss="sem.Variable@503429"
vv="sem.Variable@503429"
type="INTEGER"
isConst="false"
isLVar="true"
value="Error"
position="5"
next
INTLIT
literal="1"
type="INTEGER"
isConst="true"
isLVar="false"
value="1"
position="5"
WHILE
inLoop="false"
next="null"
loopExit="null"
position="6"
expr
stats
STATS
start="UNDEF"
loopExit="null"
inLoop="false"
position="9"
left
right
DESNULL
type="null"
isConst="false"
isLVar="false"
value="Error"
position="5"
BINARY
op="22"
type="BOOLEAN"
isConst="false"
isLVar="false"
value="Error"
position="6"
left
right
STATS
start="UNDEF"
loopExit="null"
inLoop="false"
position="7"
left
right
WRITE
inLoop="false"
next="null"
loopExit="null"
position="9"
expr
STATS
start="UNDEF"
loopExit="null"
inLoop="false"
position="9"
left
right
VARREF
ident="i"
ss="sem.Variable@503429"
vv="sem.Variable@503429"
type="INTEGER"
isConst="false"
isLVar="true"
value="Error"
position="6"
next
INTLIT
literal="10"
type="INTEGER"
isConst="true"
isLVar="false"
value="10"
position="6"
ASSIGN
inLoop="false"
next="null"
loopExit="null"
position="7"
left
right
STATNULL
left="null"
right="null"
start="UNDEF"
loopExit="null"
inLoop="false"
position="8"
BINARY
op="2"
type="INTEGER"
isConst="false"
isLVar="false"
value="Error"
position="9"
left
right
WRITELN
inLoop="false"
next="null"
loopExit="null"
position="9"
STATNULL
left="null"
right="null"
start="UNDEF"
loopExit="null"
inLoop="false"
position="10"
DESNULL
type="null"
isConst="false"
isLVar="false"
value="Error"
position="6"
VARREF
ident="i"
ss="sem.Variable@503429"
vv="sem.Variable@503429"
type="INTEGER"
isConst="false"
isLVar="true"
value="Error"
position="7"
next
BINARY
op="1"
type="INTEGER"
isConst="false"
isLVar="false"
value="Error"
position="7"
left
right
VARREF
ident="i"
ss="sem.Variable@503429"
vv="sem.Variable@503429"
type="INTEGER"
isConst="false"
isLVar="true"
value="Error"
position="9"
next
INTLIT
literal="10"
type="INTEGER"
isConst="true"
isLVar="false"
value="10"
position="9"
DESNULL
type="null"
isConst="false"
isLVar="false"
value="Error"
position="7"
VARREF
ident="i"
ss="sem.Variable@503429"
vv="sem.Variable@503429"
type="INTEGER"
isConst="false"
isLVar="true"
value="Error"
position="7"
next
INTLIT
literal="1"
type="INTEGER"
isConst="true"
isLVar="false"
value="1"
position="7"
DESNULL
type="null"
isConst="false"
isLVar="false"
value="Error"
position="9"
DESNULL
type="null"
isConst="false"
isLVar="false"
value="Error"
position="7"

Turing Machines



Turing Machines

Definition:

Turing Machines

Definition:

- Start with a finite directed graph G.

Turing Machines

Definition:

- Start with a finite directed graph G.
- Add a data structure — an infinite tape.

Turing Machines

Definition:

- Start with a finite directed graph G.
- Add a data structure — an infinite tape.

Turing Machines

Definition:

- Start with a finite directed graph G.
- Add a data structure — an infinite tape.
  - An array that can be "grown" in either direction.

Turing Machines

Definition:

- Start with a finite directed graph G.
- Add a data structure — an infinite tape.
  - An array that can be "grown" in either direction.
  - We can only inspect the array element the read/write head covers.

Turing Machines

Definition:

- Start with a finite directed graph G.
- Add a data structure — an infinite tape.
  - An array that can be "grown" in either direction.
  - We can only inspect the array element the read/write head covers.
  - Operations: read cell, write cell, and move read/write head left/right one cell.

Turing Machines

Definition:

- Start with a finite directed graph G.
- Add a data structure — an infinite tape.
  - An array that can be "grown" in either direction.
  - We can only inspect the array element the read/write head covers.
  - Operations: read cell, write cell, and move read/write head left/right one cell.
- Decorate edges of G with instructions for:

Turing Machines

Definition:

- Start with a finite directed graph G.
- Add a data structure — an infinite tape.
  - An array that can be "grown" in either direction.
  - We can only inspect the array element the read/write head covers.
  - Operations: read cell, write cell, and move read/write head left/right one cell.
- Decorate edges of G with instructions for:

Turing Machines

Definition:

- Start with a finite directed graph G.
- Add a data structure — an infinite tape.
  - An array that can be "grown" in either direction.
  - We can only inspect the array element the read/write head covers.
  - Operations: read cell, write cell, and move read/write head left/right one cell.
- Decorate edges of G with instructions for:
  - The symbol observed on the tape.

Turing Machines

Definition:

- Start with a finite directed graph G.
- Add a data structure — an infinite tape.
  - An array that can be "grown" in either direction.
  - We can only inspect the array element the read/write head covers.
  - Operations: read cell, write cell, and move read/write head left/right one cell.
- Decorate edges of G with instructions for:
  - The symbol observed on the tape.
  - Instructions for what to write to the tape.

Turing Machines

Definition:

- Start with a finite directed graph G.
- Add a data structure — an infinite tape.
  - An array that can be "grown" in either direction.
  - We can only inspect the array element the read/write head covers.
  - Operations: read cell, write cell, and move read/write head left/right one cell.
- Decorate edges of G with instructions for:
  - The symbol observed on the tape.
  - Instructions for what to write to the tape.
  - Instructions for which way to move the read/write head.

Turing Machines

DFA's are strictly weaker than CFG's, which are strictly weaker than Turing Machines.

Turing Machines

DFA's are strictly weaker than CFG's, which are strictly weaker than Turing Machines.

Compilers Viewpoint:

Turing Machines

DFA's are strictly weaker than CFG's, which are strictly weaker than Turing Machines.

Compilers Viewpoint:

- A compiler is a "proof" that the input language is no stronger than the output language.

Turing Machines

DFA's are strictly weaker than CFG's, which are strictly weaker than Turing Machines.

Compilers Viewpoint:

- A compiler is a "proof" that the input language is no stronger than the output language.
- Translating from a high-level language directly to a low-level language is hard.

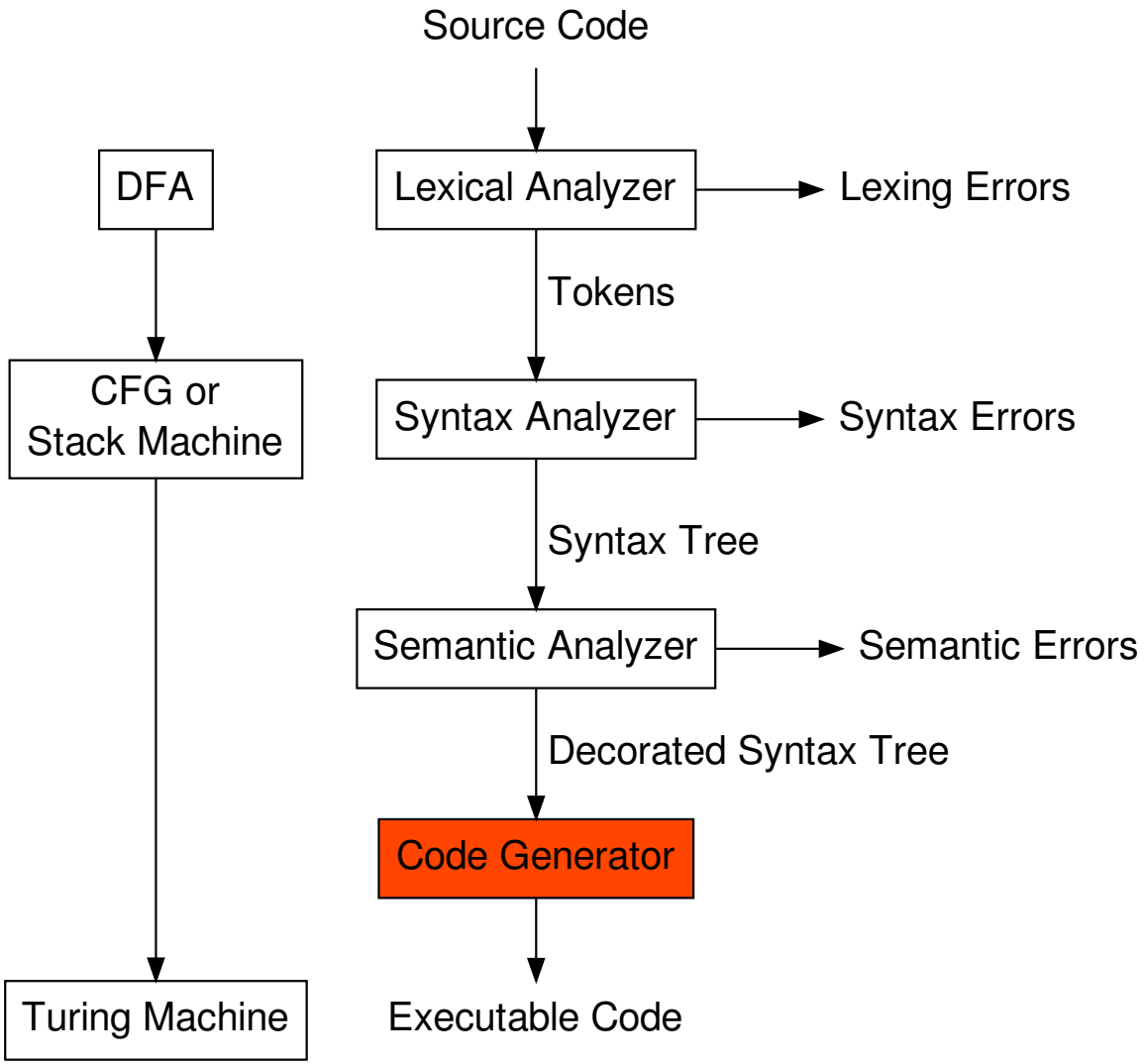
Turing Machines

DFA's are strictly weaker than CFG's, which are strictly weaker than Turing Machines.

Compilers Viewpoint:

- A compiler is a "proof" that the input language is no stronger than the output language.
- Translating from a high-level language directly to a low-level language is hard.
- Instead, we can pass through a number of intermediate languages.

Generating Machine Code



Generating Machine Code

This step is closely tied to the **architecture** or **virtual machine** you choose to target.

For Microprocessors:

Generating Machine Code

This step is closely tied to the **architecture** or **virtual machine** you choose to target.

For Microprocessors:

- Compiler Output: Instructions for the CPU

Generating Machine Code

This step is closely tied to the **architecture** or **virtual machine** you choose to target.

For Microprocessors:

- Compiler Output: Instructions for the CPU
- To perform any calculation, you need to store data in one of *finitely many* registers.

Generating Machine Code

This step is closely tied to the **architecture** or **virtual machine** you choose to target.

For Microprocessors:

- Compiler Output: Instructions for the CPU
- To perform any calculation, you need to store data in one of *finitely many* registers.
- Problem: How can we ensure:

Generating Machine Code

This step is closely tied to the **architecture** or **virtual machine** you choose to target.

For Microprocessors:

- Compiler Output: Instructions for the CPU
- To perform any calculation, you need to store data in one of *finitely many* registers.
- Problem: How can we ensure:

Generating Machine Code

This step is closely tied to the **architecture** or **virtual machine** you choose to target.

For Microprocessors:

- Compiler Output: Instructions for the CPU
- To perform any calculation, you need to store data in one of *finitely many* registers.
- Problem: How can we ensure:
  - Correctness — no data gets clobbered.

Generating Machine Code

This step is closely tied to the **architecture** or **virtual machine** you choose to target.

For Microprocessors:

- Compiler Output: Instructions for the CPU
- To perform any calculation, you need to store data in one of *finitely many* registers.
- Problem: How can we ensure:
  - Correctness — no data gets clobbered.
  - Speed

Generating Machine Code

This step is closely tied to the **architecture** or **virtual machine** you choose to target.

For Microprocessors:

- Compiler Output: Instructions for the CPU
- To perform any calculation, you need to store data in one of *finitely many* registers.
- Problem: How can we ensure:
  - Correctness — no data gets clobbered.
  - Speed
- Pros: Programs run faster.

Generating Machine Code

This step is closely tied to the **architecture** or **virtual machine** you choose to target.

For Microprocessors:

- Compiler Output: Instructions for the CPU
- To perform any calculation, you need to store data in one of *finitely many* registers.
- Problem: How can we ensure:
  - Correctness — no data gets clobbered.
  - Speed
- Pros: Programs run faster.
- Cons:

Generating Machine Code

This step is closely tied to the **architecture** or **virtual machine** you choose to target.

For Microprocessors:

- Compiler Output: Instructions for the CPU
- To perform any calculation, you need to store data in one of *finitely many* registers.
- Problem: How can we ensure:
  - Correctness — no data gets clobbered.
  - Speed
- Pros: Programs run faster.
- Cons:

Generating Machine Code

This step is closely tied to the **architecture** or **virtual machine** you choose to target.

For Microprocessors:

- Compiler Output: Instructions for the CPU
- To perform any calculation, you need to store data in one of *finitely many* registers.
- Problem: How can we ensure:
  - Correctness — no data gets clobbered.
  - Speed
- Pros: Programs run faster.
- Cons:
  - An intermediate representation is necessary.

Generating Machine Code

This step is closely tied to the **architecture** or **virtual machine** you choose to target.

For Microprocessors:

- Compiler Output: Instructions for the CPU
- To perform any calculation, you need to store data in one of *finitely many* registers.
- Problem: How can we ensure:
  - Correctness — no data gets clobbered.
  - Speed
- Pros: Programs run faster.
- Cons:
  - An intermediate representation is necessary.
  - Your compiler is tied to a particular architecture.

Example

```
PROGRAM FACTORIAL;

VAR i : INTEGER;
VAR prod : INTEGER;
BEGIN
    i := 1;
    prod := 1;
    WHILE i <= 10 DO
        prod := prod * i;
        i := i + 1;
    ENDDO;
    WRITE prod; WRITELN;
END.
```

Intermediate Representation Example

```
 1 | assign   v0  1
 2 | assign   v1  1
 3 | le       v2  v0  10
 4 | joz      v2  10
 5 | mul      v3  v1  v0
 6 | assign   v1  v3
 7 | add      v4  v0  1
 8 | assign   v0  v4
 9 | jump     3
10 | assign   v5  v1
11 | write    v5
12 | writeln
13 | end
```

Assembly Language Example (MIPS)

	Triple Code				MIPS Assembly			
1	assign	v0	1		li	$t0,	1	
					sw	$t0,	24($sp)	
2	assign	v1	1		li	$t0,	1	
					sw	$t0,	28($sp)	
3	le	v2	v0	10	lw	$t1,	24($sp)	
					li	$t2,	10	
					sle	$t0,	$t1,	$t2
					sw	$t0,	40($sp)	
4	joz	v2	10		label_3:			
					lw	$t0,	40($sp)	
					beq	$zero,	$t0,	label_10

Virtual Machines / Interpreters

Virtual Machines / Interpreters

- Compiler Output: A sequence of bytes to be interpreted as low-level instructions by another program.

Virtual Machines / Interpreters

- Compiler Output: A sequence of bytes to be interpreted as low-level instructions by another program.
- Pros:

Virtual Machines / Interpreters

- Compiler Output: A sequence of bytes to be interpreted as low-level instructions by another program.
- Pros:

Virtual Machines / Interpreters

- Compiler Output: A sequence of bytes to be interpreted as low-level instructions by another program.
- Pros:
  - Easier to produce virtual machine code.

Virtual Machines / Interpreters

- Compiler Output: A sequence of bytes to be interpreted as low-level instructions by another program.
- Pros:
  - Easier to produce virtual machine code.
  - Easier to implement “high level features” correctly.

Virtual Machines / Interpreters

- Compiler Output: A sequence of bytes to be interpreted as low-level instructions by another program.
- Pros:
  - Easier to produce virtual machine code.
  - Easier to implement “high level features” correctly.
  - Not tied to a particular architecture.

Virtual Machines / Interpreters

- Compiler Output: A sequence of bytes to be interpreted as low-level instructions by another program.
- Pros:
  - Easier to produce virtual machine code.
  - Easier to implement "high level features" correctly.
  - Not tied to a particular architecture.
- Cons: Poor performance (sometimes).

Virtual Machines / Interpreters

Many languages use a virtual machine or interpreter rather than compiling to assembly:

Virtual Machines / Interpreters

Many languages use a virtual machine or interpreter rather than compiling to assembly:

- Examples: Java, Python, Lisp

Virtual Machines / Interpreters

Many languages use a virtual machine or interpreter rather than compiling to assembly:

- Examples: Java, Python, Lisp
- Examples: Sage, GAP

Virtual Machines / Interpreters

Many languages use a virtual machine or interpreter rather than compiling to assembly:

- Examples: Java, Python, Lisp
- Examples: Sage, GAP
- Many interpreters are stack-based. Bytecode running on a virtual machine looks like:

Virtual Machines / Interpreters

Many languages use a virtual machine or interpreter rather than compiling to assembly:

- Examples: Java, Python, Lisp
- Examples: Sage, GAP
- Many interpreters are stack-based. Bytecode running on a virtual machine looks like:

Virtual Machines / Interpreters

Many languages use a virtual machine or interpreter rather than compiling to assembly:

- Examples: Java, Python, Lisp
- Examples: Sage, GAP
- Many interpreters are stack-based. Bytecode running on a virtual machine looks like:
  - A list of instructions.

Virtual Machines / Interpreters

Many languages use a virtual machine or interpreter rather than compiling to assembly:

- Examples: Java, Python, Lisp
- Examples: Sage, GAP
- Many interpreters are stack-based. Bytecode running on a virtual machine looks like:
  - A list of instructions.
  - An array of memory (for storing variables).

Virtual Machines / Interpreters

Many languages use a virtual machine or interpreter rather than compiling to assembly:

- Examples: Java, Python, Lisp
- Examples: Sage, GAP
- Many interpreters are stack-based. Bytecode running on a virtual machine looks like:
  - A list of instructions.
  - An array of memory (for storing variables).
  - A stack for performing calculations (in lieu of registers).

Virtual Machines / Interpreters

Many virtual machines use an **operand stack** to calculate.

Virtual Machines / Interpreters

Many virtual machines use an **operand stack** to calculate.

- Instead of operating on registers:

Virtual Machines / Interpreters

Many virtual machines use an **operand stack** to calculate.

- Instead of operating on registers:

Virtual Machines / Interpreters

Many virtual machines use an **operand stack** to calculate.

- Instead of operating on registers:
  - Push values onto the stack.

Virtual Machines / Interpreters

Many virtual machines use an **operand stack** to calculate.

- Instead of operating on registers:
  - Push values onto the stack.
  - Pop values off, operate on them.

Virtual Machines / Interpreters

Many virtual machines use an **operand stack** to calculate.

- Instead of operating on registers:
  - Push values onto the stack.
  - Pop values off, operate on them.
  - Push the results onto the stack.

Virtual Machines / Interpreters

Many virtual machines use an **operand stack** to calculate.

- Instead of operating on registers:
  - Push values onto the stack.
  - Pop values off, operate on them.
  - Push the results onto the stack.
- Similar to Reverse Polish Notation.

Virtual Machines / Interpreters

Many virtual machines use an **operand stack** to calculate.

- Instead of operating on registers:
  - Push values onto the stack.
  - Pop values off, operate on them.
  - Push the results onto the stack.
- Similar to Reverse Polish Notation.

Virtual Machines / Interpreters

Many virtual machines use an **operand stack** to calculate.

- Instead of operating on registers:
  - Push values onto the stack.
  - Pop values off, operate on them.
  - Push the results onto the stack.
- Similar to Reverse Polish Notation.

Example: `5 * (3 + 2)` becomes

Instructions	Stack State
`push 5`	5
`push 3`	5, 3
`push 2`	5, 3, 2
`add`	5, (3+2)
`mul`	5(3+2)

Bytecode Example

```
 0 | alloc  2                    || 16 | get
 1 | push   0   % Addr.  i       || 17 | mul_i
 2 | push   1   % Value := 1     || 18 | store
 3 | store                       || 19 | push     0   % Addr.  i
 4 | push   1   % Addr.  prod    || 20 | push     0   % Addr.  i
 5 | push   1   % Value := 1     || 21 | get
 6 | store                       || 22 | push     1   % Value := 1
 7 | push   0   % WHILE start    || 23 | add_i
 8 | get                         || 24 | store
 9 | push   10  % Value := 10    || 25 | jump     7   % WHILE end
10 | le                          || 26 | push     1   % Addr.  prod
11 | joz    26                   || 27 | get
12 | push   1   % Addr.  prod    || 28 | write_I
13 | push   1   % Addr.  prod    || 29 | writeln
14 | get                         || 30 | end
15 | push   0   % Addr.  i       ||    |
```

Open Problems

Open Problems

1. Obfuscation : Given a program P, how can I modify P to make it difficult for others to steal my trade secrets, algorithms, etc.?

Open Problems

1. Obfuscation : Given a program P, how can I modify P to make it difficult for others to steal my trade secrets, algorithms, etc.?
2. Deobfuscation : Given an obfuscated program P, what can I learn about the operations of P?

Open Problems

1. Obfuscation : Given a program P, how can I modify P to make it difficult for others to steal my trade secrets, algorithms, etc.?
2. Deobfuscation : Given an obfuscated program P, what can I learn about the operations of P?
3. Decompilation : Given an executable E, can I generate source code S that compiles to E?