Languages by Humans for Computers

Our last topic in this overview of compilation is **language grammars**. These are the rules used to define programming languages, and we'll see how they provide enough structure to permit *efficient and unambiguous* translation from program **source code** to machine code.

We won't have time to dive too deeply into grammars, parsing, and lexical analysis; but we'll be able to see and understand how the translation process from source to machine code is performed.

Ambiguity is Clearly a Bad Thing

Mathematics is an **unambiguous** language, given an expression:

$$x^{z+2} + 100 + y + z(2 + ab)$$

It has **only one value** (for a collection of variable values)

Ambiguity is Clearly a Bad Thing

Mathematics is an **unambiguous** language, given an expression:

$$x^{z+2} + 100 + y + z(2 + ab)$$

It has **only one value** (for a collection of variable values)

Programming languages may permit **more than one way** to express results or outcomes…

`print("Hello world")`	Python	`sys.stdout.write("Hello world\n")`
`SELECT name FROM students;`	SQL	`SELECT s.name FROM students AS s;`
`x = x + 1;`	C/C++	`++x;`

There is still only one way to interpret the **intent of the programmer**.

Defining a Language

Here is a simple grammar that defines a language:

$$
\begin{array}{lcl}
S & \rightarrow & A\ \$\ |\ x\ B\ x\ \$ \\
A & \rightarrow & s\ B\ t\ |\ w \\
B & \rightarrow & q\ s\ |\ s\ q
\end{array}
$$

By Convention...

1. the special symbol \$ means the **end of input**
2. UPPER case terms are **non-terminals**, they can appear on either side of the $\rightarrow$
3. terms *other than \$ and non-terminals* are called **terminals**, they can appear only on the right-hand side of $\rightarrow$
4. the vertical bar, $|$, is read as “OR”

Programs consist of *only* terminals.

Defining a Language

A simple grammar that defines a language:

$$\begin{array}{lcl} S & \rightarrow & A\ \$ \mid x\ B\ x\ \$ \\ A & \rightarrow & s\ B\ t \mid w \\ B & \rightarrow & q\ s \mid s\ q \end{array}$$

Is the single token w permitted by this grammar?

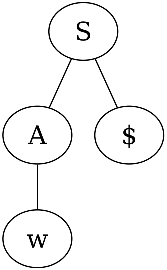
Defining a Language

A simple grammar that defines a language:

$$\begin{array}{lcl} S & \rightarrow & A\ \$ \mid x\ B\ x\ \$ \\ A & \rightarrow & s\ B\ t \mid w \\ B & \rightarrow & q\ s \mid s\ q \end{array}$$

Is the single token w permitted by this grammar? **Yes**

$$\begin{array}{lcl} S & \rightarrow & A\ \$ \\ S & \rightarrow & w\ \$ \end{array}$$



←Parse Tree!

Defining a Language

A simple grammar that defines a language:

$$
\begin{array}{lcl}
S & \rightarrow & A\ \$\ |\ x\ B\ x\ \$ \\
A & \rightarrow & s\ B\ t\ |\ w \\
B & \rightarrow & q\ s\ |\ s\ q
\end{array}
$$

What about $x\ s\ q\ x$?

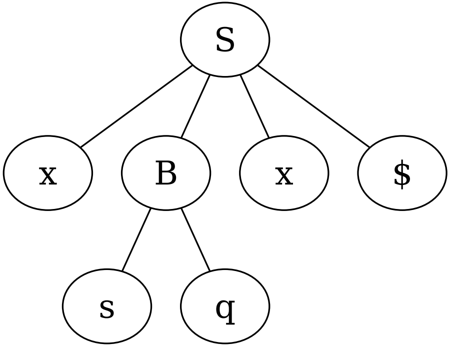
Defining a Language

A simple grammar that defines a language:

$$
\begin{aligned}
S &\rightarrow A\ \$\ |\ x\ B\ x\ \$ \\
A &\rightarrow s\ B\ t\ |\ w \\
B &\rightarrow q\ s\ |\ s\ q
\end{aligned}
$$

What about $x\ s\ q\ x$? **Yes**

$$
\begin{aligned}
S &\rightarrow x\ B\ x\ \$ \\
S &\rightarrow x\ s\ q\ x\ \$
\end{aligned}
$$



Defining a Language

A simple grammar that defines a language:

$$
\begin{array}{lcl}
S & \rightarrow & A\ \$ \mid x\ B\ x\ \$ \\
A & \rightarrow & s\ B\ t \mid w \\
B & \rightarrow & q\ s \mid s\ q
\end{array}
$$

Which of these are permitted?

$s\ s\ q\ t$

$w\ x\ q\ s\ x$

$x\ q\ s\ x$

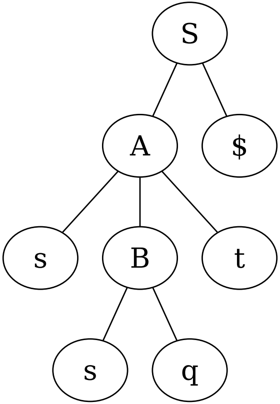
Defining a Language

A simple grammar that defines a language:

$$
\begin{array}{lcl}
S & \rightarrow & A \ \$ \mid x \ B \ x \ \$ \\
A & \rightarrow & s \ B \ t \mid w \\
B & \rightarrow & q \ s \mid s \ q
\end{array}
$$

Which of these are permitted?

$s \ s \ q \ t$ **Yes**



Defining a Language

A simple grammar that defines a language:

$$\begin{aligned} S &\rightarrow A\ \$ \mid x\ B\ x\ \$ \\ A &\rightarrow s\ B\ t \mid w \\ B &\rightarrow q\ s \mid s\ q \end{aligned}$$

Which of these are permitted?

$w\ x\ q\ s\ x$?

Defining a Language

A simple grammar that defines a language:

$$
\begin{array}{lcl}
S & \rightarrow & A\ \$ \mid x\ B\ x\ \$ \\
A & \rightarrow & s\ B\ t \mid w \\
B & \rightarrow & q\ s \mid s\ q
\end{array}
$$

Which of these are permitted?

Syntax Error: did not expect x after w.

$w\ x\ q\ s\ x$ No

Defining a Language

$$\begin{array}{lcl} S & \rightarrow & A\ \$\ \mid\ x\ B\ x\ \$ \\ A & \rightarrow & s\ B\ t\ \mid\ w \\ B & \rightarrow & q\ s\ \mid\ s\ q \end{array}$$

Which of these are permitted?

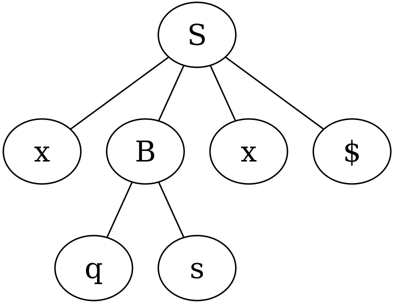
$x\ q\ s\ x$?

Defining a Language

$$\begin{array}{lcl} S & \rightarrow & A\ \$\ \mid\ x\ B\ x\ \$ \\ A & \rightarrow & s\ B\ t\ \mid\ w \\ B & \rightarrow & q\ s\ \mid\ s\ q \end{array}$$

Which of these are permitted?

$x\ q\ s\ x$ **Yes**

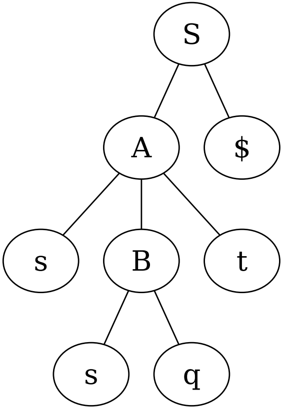


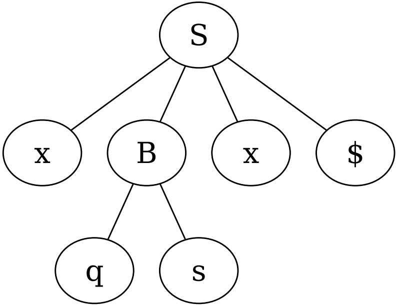
Defining a Language

a. The parse tree **leaves** are special, what do they hold?

b. What grammar parts are in the **non-leaf** nodes?

$$
\begin{array}{lcl}
S & \rightarrow & A\ \$ \mid x\ B\ x\ \$ \\
A & \rightarrow & s\ B\ t \mid w \\
B & \rightarrow & q\ s \mid s\ q
\end{array}
$$





Defining a Language

```
w
s q s t
s s q t
x q s x
x s q x
```

$$
\begin{array}{lcl}
S & \rightarrow & A\ \$\ \mid\ x\ B\ x\ \$ \\
A & \rightarrow & s\ B\ t\ \mid\ w \\
B & \rightarrow & q\ s\ \mid\ s\ q
\end{array}
$$

Clearly, there are a limited number of **terminal sequences** permitted by this grammar.

It is FINITE — we certainly don't want programming languages with this property.

Recursive Language Definitions

$$
\begin{array}{rcl}
S & \rightarrow & QLIST\ \$ \\
QLIST & \rightarrow & Q\ QLIST \\
 & | & \lambda \\
Q & \rightarrow & a\ b\ c \\
 & | & k \\
 & | & s\ t\ u
\end{array}
$$

Here is a *recursive* grammar that permits an infinite collection of terminal sequences.

One Last Convention...

5. the special symbol λ means an **empty sequence** of tokens — AKA “nothing”

Recursive Language Definitions

$$
\begin{array}{rcl}
S & \rightarrow & QLIST\ \$ \\
QLIST & \rightarrow & Q\ QLIST \\
 & | & \lambda \\
Q & \rightarrow & a\ b\ c \\
 & | & k \\
 & | & s\ t\ u
\end{array}
$$

Here is a *recursive* grammar that permits an infinite collection of terminal sequences.

What happens with multiple Qs?

$k\ a\ b\ c\ k$

Recursive Language Definitions

Program

$k\ a\ b\ c\ k$

parsed by language

$$
\begin{array}{rcl}
S & \rightarrow & QLIST\ \$ \\
QLIST & \rightarrow & Q\ QLIST \\
 & | & \lambda \\
Q & \rightarrow & a\ b\ c \\
 & | & k \\
 & | & s\ t\ u
\end{array}
$$

$\rightarrow$

Derivation

$$
\begin{array}{lcl}
S & \rightarrow & QLIST\ \$ \\
S & \rightarrow & Q\ QLIST\ \$ \\
S & \rightarrow & k\ QLIST\ \$ \\
S & \rightarrow & k\ Q\ QLIST\ \$ \\
S & \rightarrow & k\ a\ b\ c\ QLIST\ \$ \\
S & \rightarrow & k\ a\ b\ c\ Q\ QLIST\ \$ \\
S & \rightarrow & k\ a\ b\ c\ k\ QLIST\ \$ \\
S & \rightarrow & k\ a\ b\ c\ k\ \lambda\ \$
\end{array}
$$

Recursive Language Definitions

Program

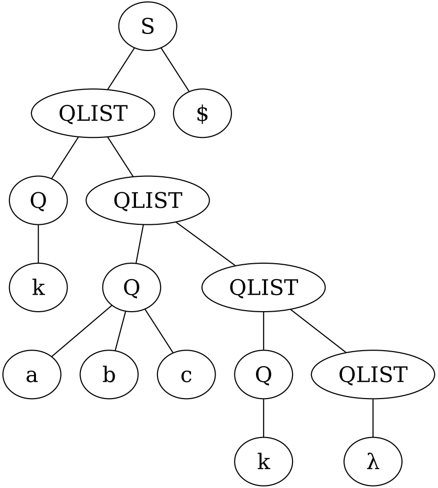
$k \;\; a \;\; b \;\; c \;\; k$

parsed by language

$$
\begin{array}{rcl}
S & \rightarrow & QLIST \;\; \$ \\
QLIST & \rightarrow & Q \;\; QLIST \\
 & | & \lambda \\
Q & \rightarrow & a \;\; b \;\; c \\
 & | & k \\
 & | & s \;\; t \;\; u
\end{array}
$$

$\rightarrow$

Parse Tree



A Simple Programming Language

$$
\begin{array}{rcl}
PROGRAM & \rightarrow & SLIST\ \$ \\
SLIST & \rightarrow & S\ SLIST \\
 & | & \lambda \\
S & \rightarrow & var\ =\ EXPR \\
 & | & if\ EXPR\ then\ (\ SLIST\ ) \\
 & | & if\ EXPR\ then\ (\ SLIST\ )\ else\ (\ SLIST\ ) \\
 & | & while\ EXPR\ do\ (\ SLIST\ ) \\
 & | & repeat\ (\ SLIST\ )\ while\ EXPR \\
 & | & repeat\ (\ SLIST\ )\ until\ EXPR \\
EXPR & \rightarrow & expr \\
 & | & var
\end{array}
$$

A Simple Programming Language

$$
\begin{array}{rcl}
PROGRAM & \rightarrow & SLIST\ \$ \\
SLIST & \rightarrow & S\ SLIST \\
 & | & \lambda \\
S & \rightarrow & var\ =\ EXPR \\
 & | & if\ EXPR\ then\ (\ SLIST\ ) \\
 & | & if\ EXPR\ then\ (\ SLIST\ )\ else\ (\ SLIST\ ) \\
 & | & while\ EXPR\ do\ (\ SLIST\ ) \\
 & | & repeat\ (\ SLIST\ )\ while\ EXPR \\
 & | & repeat\ (\ SLIST\ )\ until\ EXPR \\
EXPR & \rightarrow & expr \\
 & | & var
\end{array}
$$

Which grammar rule means this is **not** a finite language?

Program Example A - Compilation Steps

I. Lexical analysis detects **keywords**, variable names, expressions, and special symbols such as parenthesis and equal.
II. The sequence of tokens is **parsed** using the grammar rules into a parse tree.
III. The parse tree is simplified into a **sequence of assignments** with **comparisons and branches (jumps)**.
IV. Assign memory locations for variables
V. Assign registers and instructions for expressions
VI. Generate comparision and jump instructions for control structures
VII. . . . and then we can generate machine instructions!

```
a = 3
c = (27)
b = a*10+1
repeat (
  b = b-a/2+1
  a = a+1
) until b<0
```

```
a = 3
c = (27)
b = a*10+1
repeat (
  b = b-a/2+1
  a = a+1
) until b<0
```

Lexical Analysis Describes Tokens →

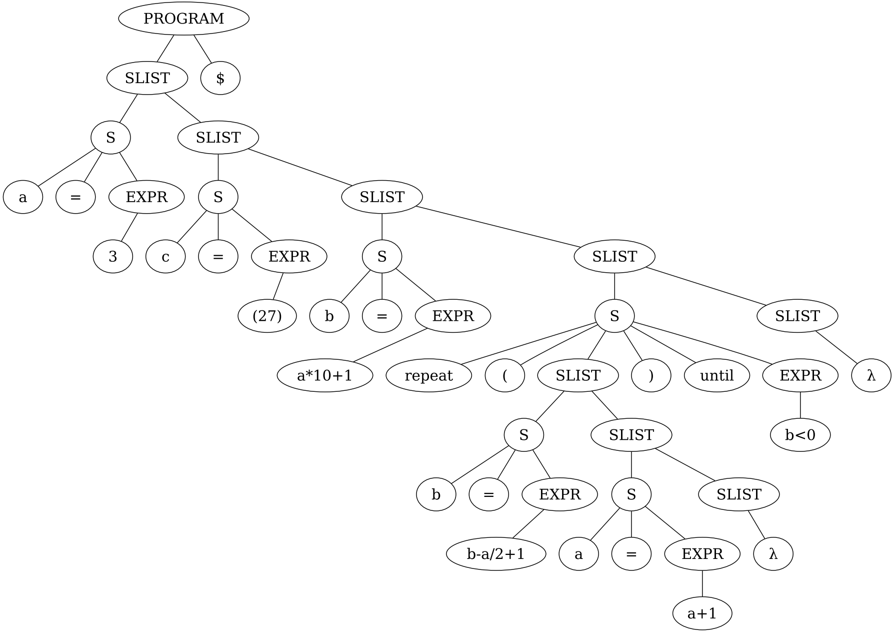
```
{var,a} = {expr,3}
{var,c} = {expr,(27)}
{var,b} = {expr,a*10+1}
repeat (
  {var,b} = {expr,b-a/2+1}
  {var,a} = {expr,a+1}
) until {expr,b<0}
```

for Grammar Analysis →

$$
\begin{array}{rcl}
PROGRAM & \rightarrow & SLIST \ \$ \\
SLIST & \rightarrow & S \ SLIST \\
 & | & \lambda \\
S & \rightarrow & var \ = \ EXPR \\
 & | & if \ EXPR \ then \ (\ SLIST \ ) \\
 & | & if \ EXPR \ then \ (\ SLIST \ ) \ else \ (\ SLIST \ ) \\
 & | & while \ EXPR \ do \ (\ SLIST \ ) \\
 & | & repeat \ (\ SLIST \ ) \ while \ EXPR \\
 & | & repeat \ (\ SLIST \ ) \ until \ EXPR \\
EXPR & \rightarrow & expr \\
 & | & var
\end{array}
$$

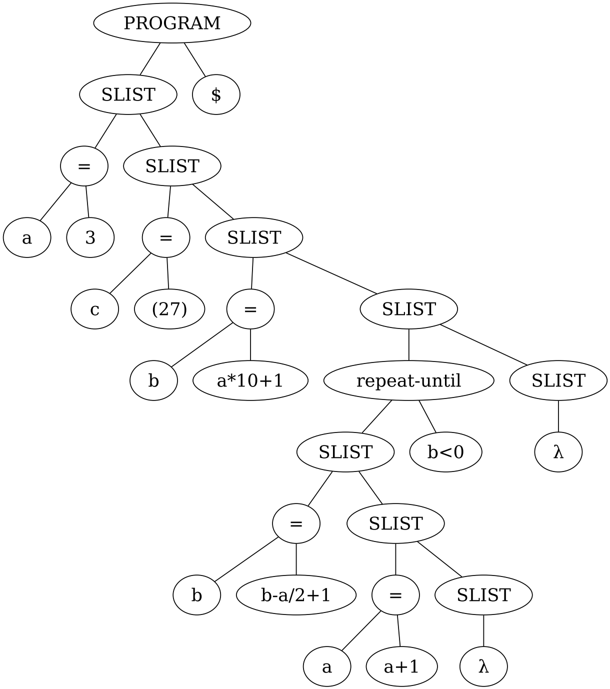
. . . that generates →

. . . a Parse Tree

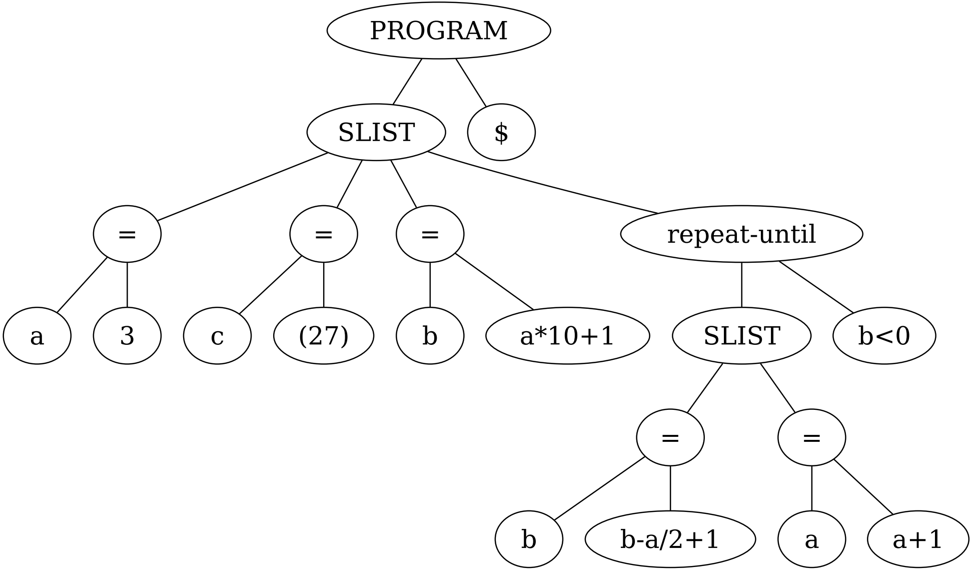


PROGRAM
SLIST
$
S
SLIST
a
=
EXPR
3
S
c
=
EXPR
(27)
SLIST
S
b
=
EXPR
a*10+1
SLIST
S
repeat
(
SLIST
)
until
EXPR
b<0
S
b
=
EXPR
b-a/2+1
SLIST
S
a
=
EXPR
a+1
SLIST
λ
SLIST
λ

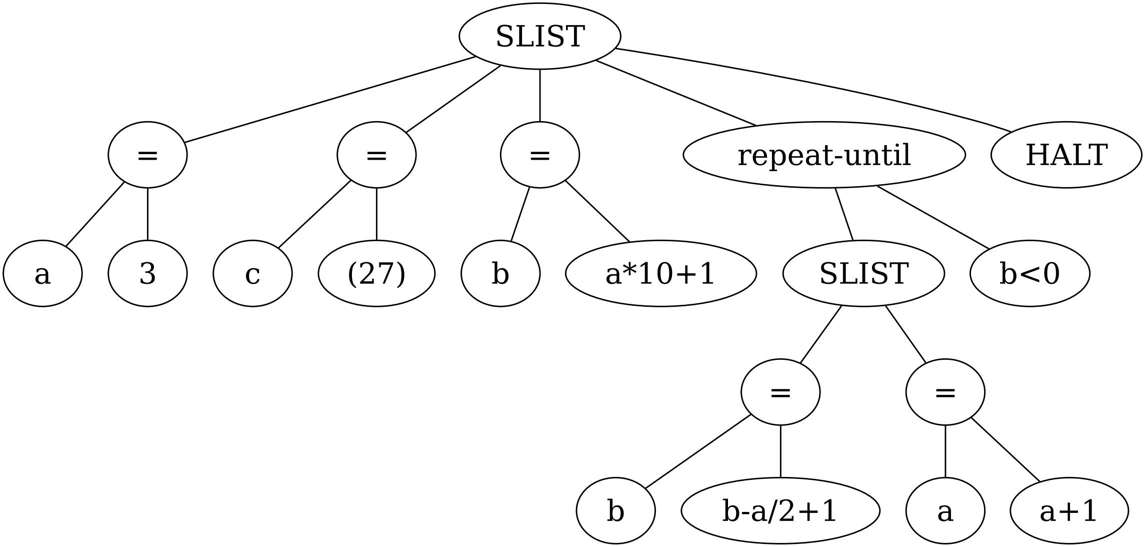
Simplify Statement *S* Nodes



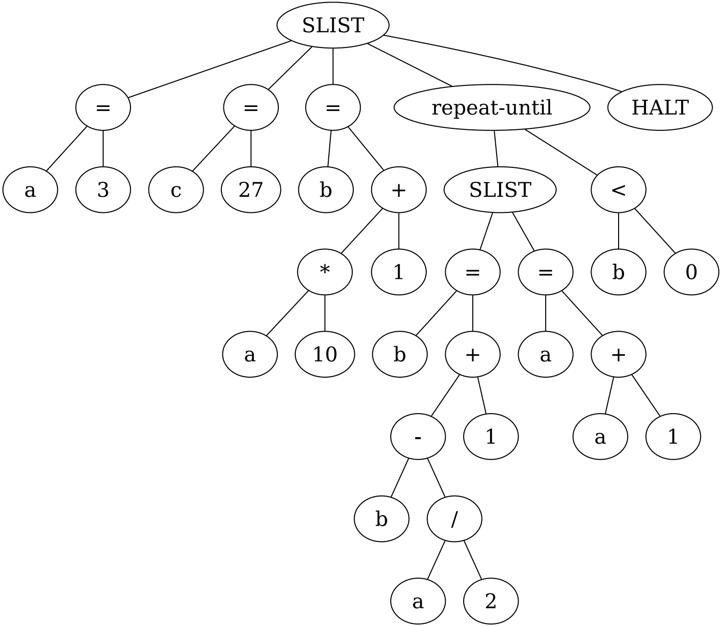
De-curse *SLIST*



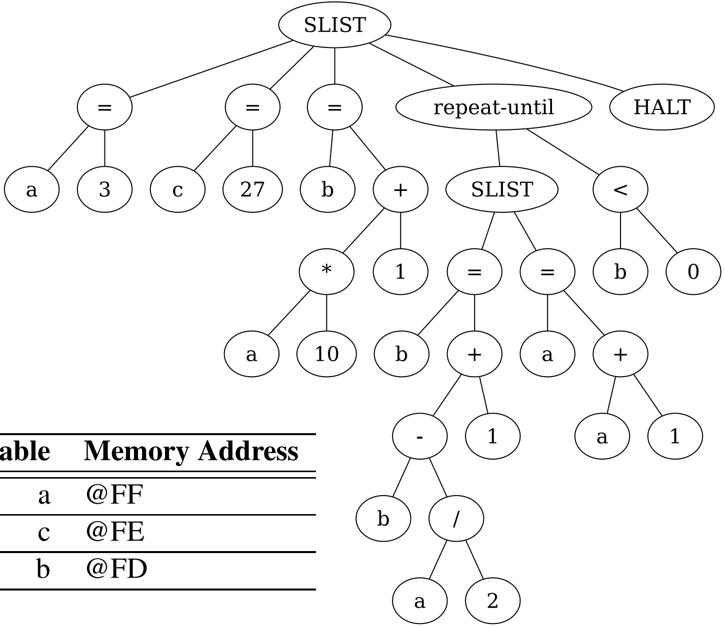
Programs are just an *SLIST* that HALTs



Generate RHS Expression Trees

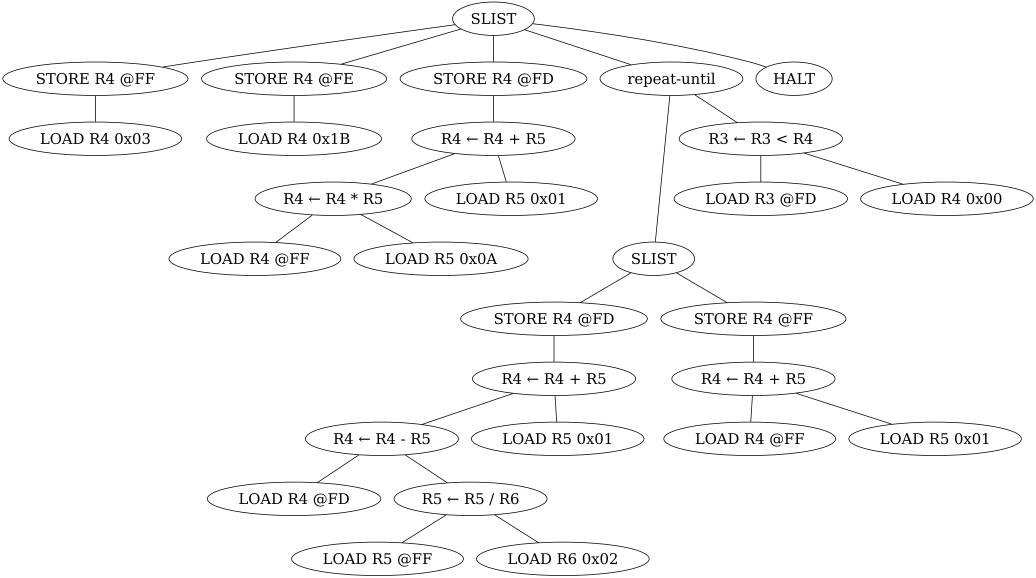


Assign Memory Locations for Variables

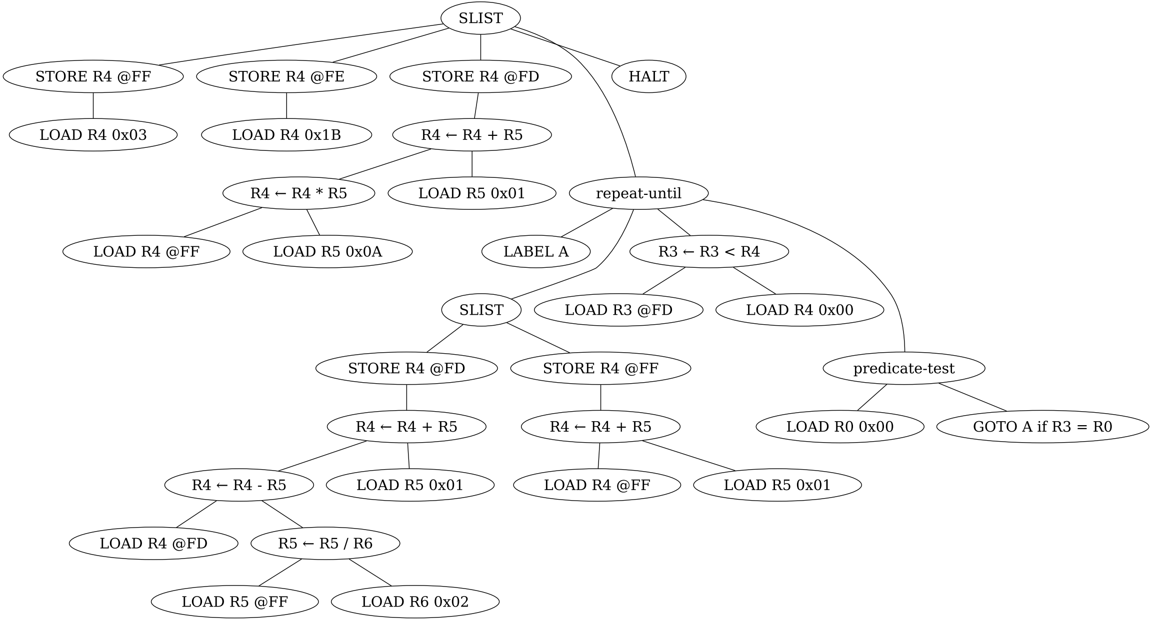


Variable	Memory Address
a	@FF
c	@FE
b	@FD

Assign Registers and Instructions for Expressions



Generate Instructions for Control Structures



(Pseudo) Assembly Code

```
LOAD R4 0x03
STORE R4 @FF
LOAD R4 0x1B
STORE R4 @FE
LOAD R4 @FF
LOAD R5 0x0A
R4 <- R4 * R5
LOAD R5 0x01
R4 <- R4 + R5
STORE R4 @FD
LABEL A
LOAD R4 @FD
LOAD R5 @FF
LOAD R6 0x02
R5 <- R5 / R6
```

```
R4 <- R4 - R5
LOAD R5 0x01
R4 <- R4 + R5
STORE R4 @FD
LOAD R4 @FF
LOAD R5 0x01
R4 <- R4 + R5
STORE R4 @FF
LOAD R3 @FD
LOAD R4 0x00
R3 <- R3 < R4
LOAD R0 0x00
GOTO A if R3 = R0
HALT
```

An **assembler** will take these instructions and generate the actual machine code, resolving **`LABEL`**s and **`GOTO`**s, and perhaps performing some low-level optimizations such as removing redundant **`LOAD`**s and **`STORE`**s.

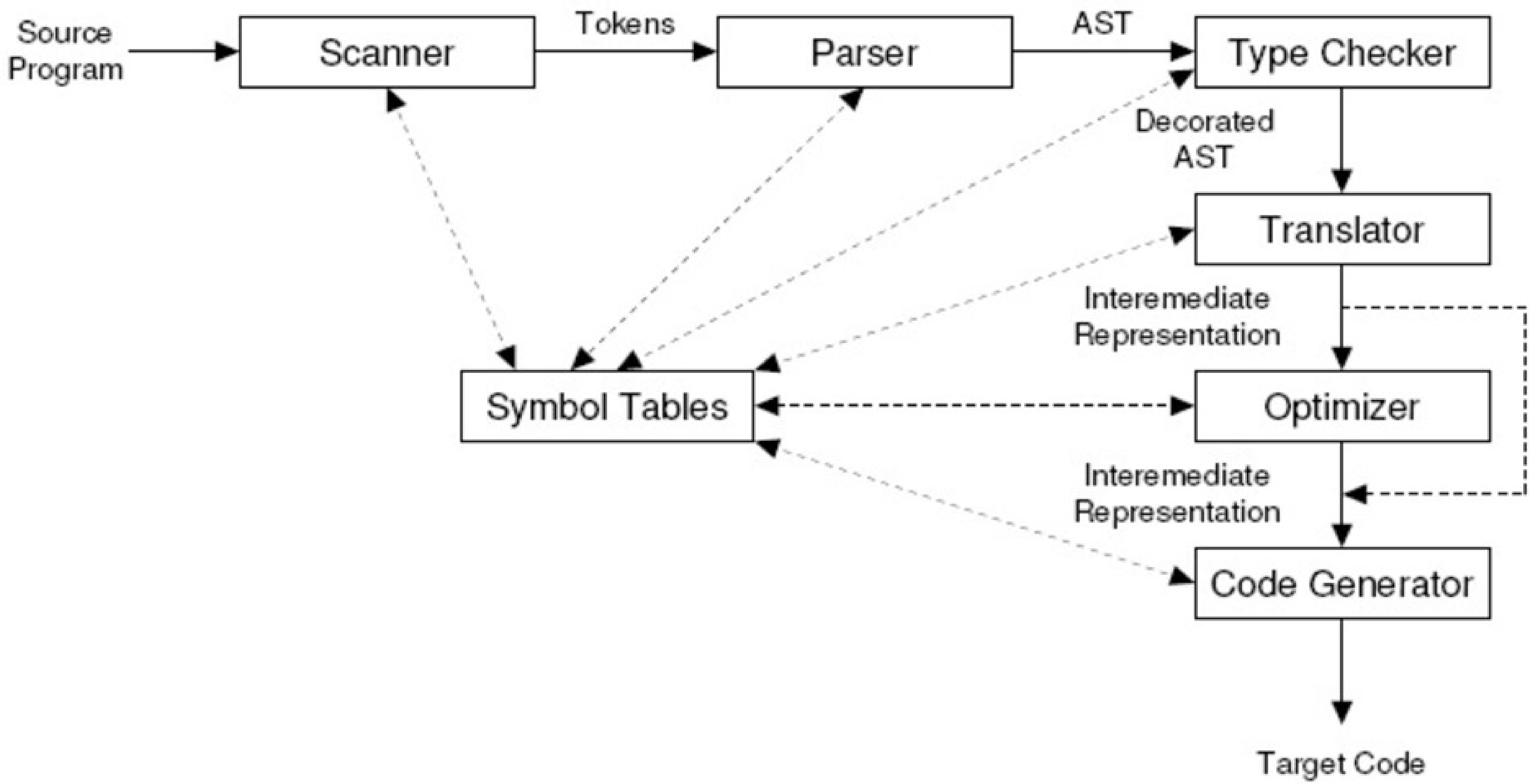


Figure 1.4: A syntax-directed compiler. AST denotes the Abstract Syntax Tree.

Compilers

We've looked at the essential steps taken to generate machine code from a high-level programming language.

Whether you use an **interpretted language** within a "virtual machine" such as Python, Ruby, Lisp, Java, . . . or a language compiled "down to machine code" such as C, C++, or Fortran — you use a compiler of some sort, and that compiler pretty much follows all of these steps in order to (eventually) execute *your instructions* on a CPU.

The design of computer languages and the parsing algorithms associated with them is one of the classic and fundamental topics in Computer Science.

Program Example B - Compilation Steps

I. Lexical analysis detects **keywords**, variable names, expressions, and special symbols such as parenthesis and equal.

II. The sequence of tokens is **parsed** using the grammar rules into a parse tree.

III. The parse tree is simplified into a **sequence of assignments** with **comparisons and branches (jumps)**.

IV. Assign memory locations for variables

V. Assign registers and instructions for expressions

VI. Generate comparision and jump instructions for control structures

VII. . . . and then we can generate machine instructions!

```
x = (a+b)/23
if x<3 then (
    s = 1
) else (
    t = 2
)
```

```
x = (a+b)/23
if x<3 then (
    s = 1
) else (
    t = 2
)
```

Lexical Analysis Describes Tokens →

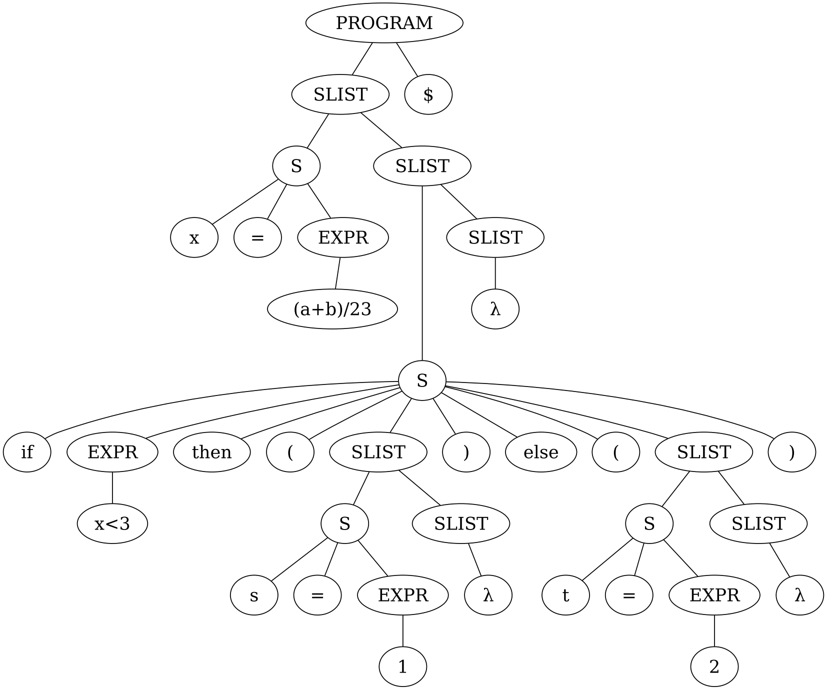
```
{var,x} = {expr,(a+b)/23}
if {expr,x<3} then (
  {var,s} = {expr,1}
) else (
  {var,t} = {expr,2}
)
```

for Grammar Analysis →

$$
\begin{array}{rcl}
\mathit{PROGRAM} & \rightarrow & \mathit{SLIST}\ \$ \\
\mathit{SLIST} & \rightarrow & \mathit{S}\ \mathit{SLIST} \\
 & | & \lambda \\
\mathit{S} & \rightarrow & \mathit{var}\ =\ \mathit{EXPR} \\
 & | & \mathit{if}\ \mathit{EXPR}\ \mathit{then}\ (\ \mathit{SLIST}\ ) \\
 & | & \mathit{if}\ \mathit{EXPR}\ \mathit{then}\ (\ \mathit{SLIST}\ )\ \mathit{else}\ (\ \mathit{SLIST}\ ) \\
 & | & \mathit{while}\ \mathit{EXPR}\ \mathit{do}\ (\ \mathit{SLIST}\ ) \\
 & | & \mathit{repeat}\ (\ \mathit{SLIST}\ )\ \mathit{while}\ \mathit{EXPR} \\
 & | & \mathit{repeat}\ (\ \mathit{SLIST}\ )\ \mathit{until}\ \mathit{EXPR} \\
\mathit{EXPR} & \rightarrow & \mathit{expr} \\
 & | & \mathit{var}
\end{array}
$$

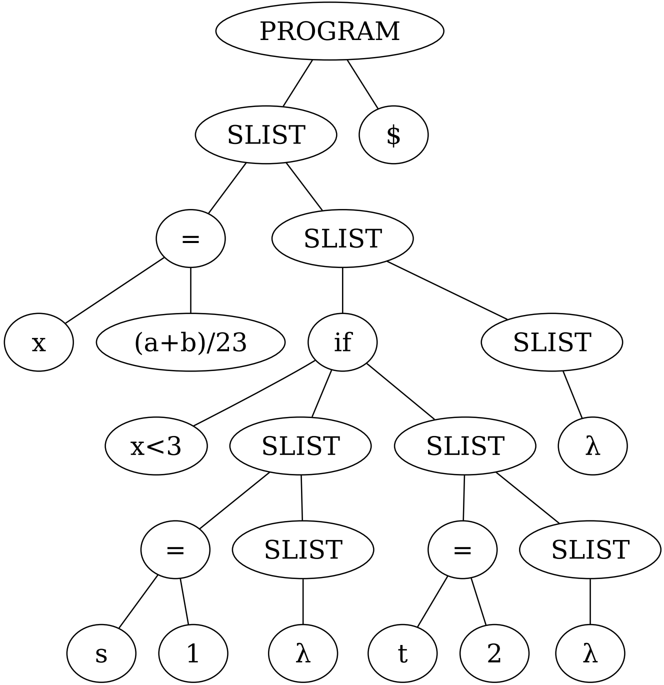
. . . that generates →

. . . a Parse Tree

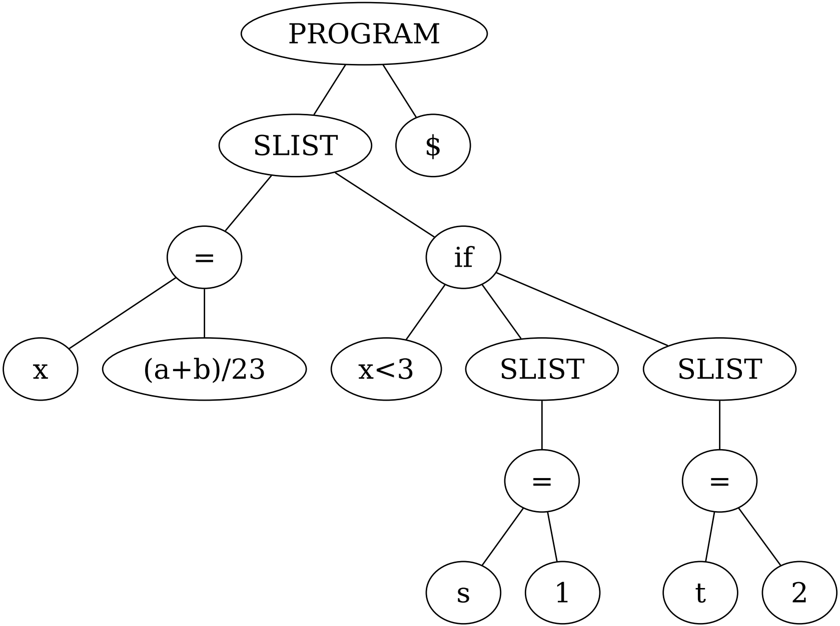


PROGRAM
SLIST
$
S
SLIST
x
=
EXPR
SLIST
(a+b)/23
λ
S
if
EXPR
then
(
SLIST
)
else
(
SLIST
)
x<3
S
SLIST
S
SLIST
s
=
EXPR
λ
t
=
EXPR
λ
1
2

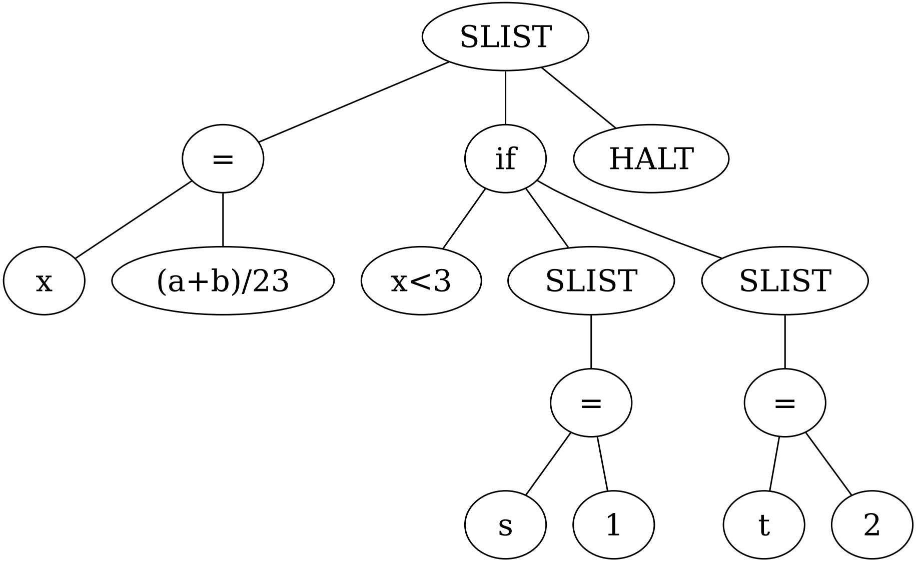
Simplify Statement S Nodes



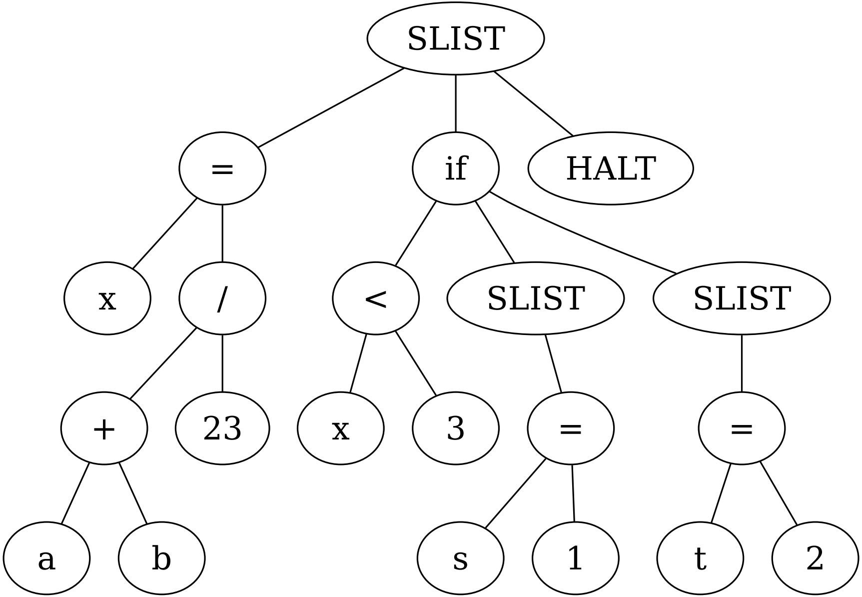
De-curse *SLIST*



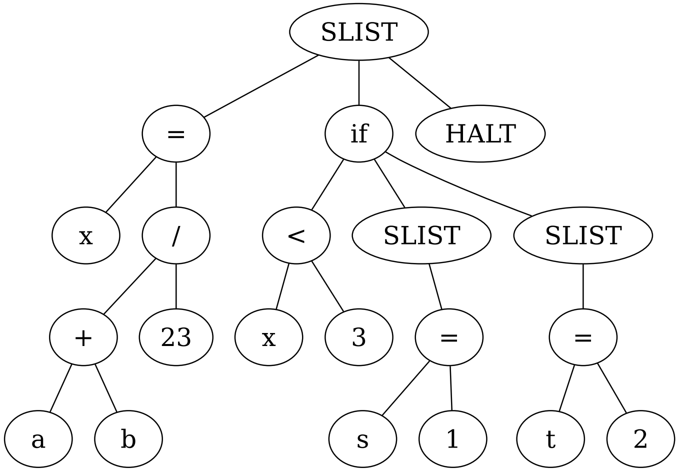
Programs are just an *SLIST* that HALTs



Generate RHS Expression Trees

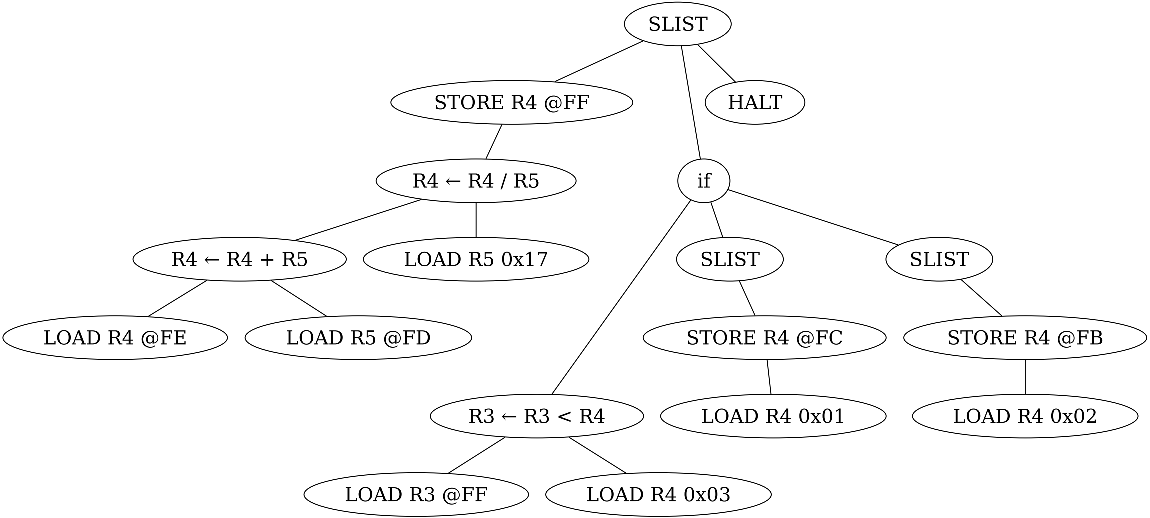


Assign Memory Locations for Variables

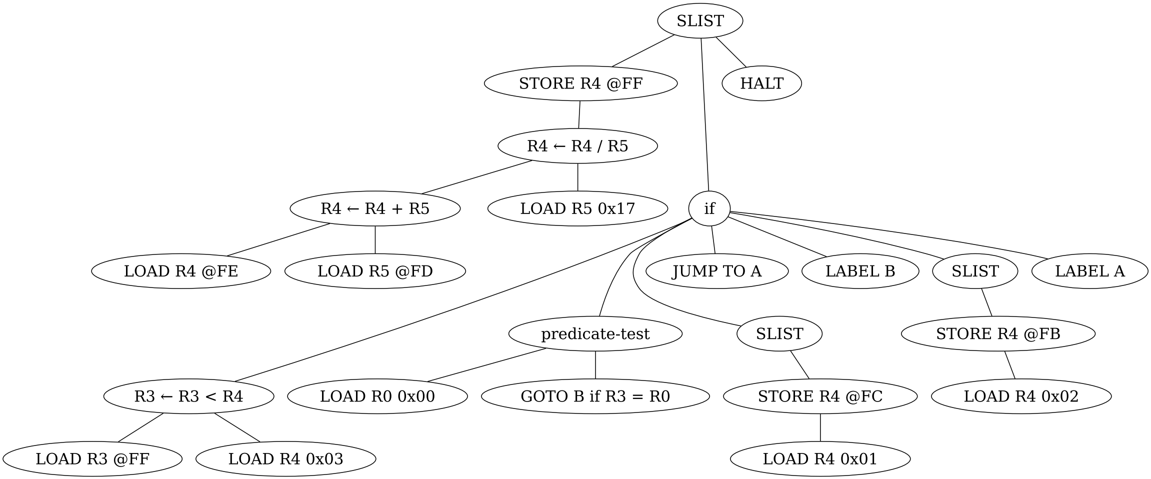


Variable	Memory Address
x	@FF
a	@FE
b	@FD
s	@FC
t	@FB

Assign Registers and Instructions for Expressions



Generate Instructions for Control Structures



(Pseudo) Assembly Code

```
LOAD R4 @FE
LOAD R5 @FD
R4 <- R4 + R5
LOAD R5 0x17
R4 <- R4 / R5
STORE R4 @FF
LOAD R3 @FF
LOAD R4 0x03
R3 <- R3 < R4
LOAD R0 0x00
GOTO B if R3 = R0
LOAD R4 0x01
STORE R4 @FC
JUMP TO A
LABEL B
LOAD R4 0x02
STORE R4 @FB
LABEL A
HALT
```

An **assembler** will take these instructions and generate the actual machine code, resolving **`LABEL`**s and **`GOTO`**s, and perhaps performing some low-level optimizations such as removing redundant **`LOAD`**s and **`STORE`**s.