

Compiler Design Project Report

Designing a Suzuka compiler using python



Done By: -

AP20110010625 – Sarika Sumanth

AP20110010598 – K. Purna Sai

AP20110010582 – Chiranjeevi Barik

AP20110010597 – M. Teja Varma

AP20110010600 – B. Sri Ram Chowdary

AP20110010566 – A. Sai Manikanta

AP20110010574 – D.S. Aditya Vasanth

AP20110010599 – Shaik Riyaz

**Description of our compiler**

We developed Suzuka as a general-purpose and mini computer programming language which can be used in many domains.

Here, we tried to simplify the data types and variables so that it will be easier for the user to get familiar with the compiler easily.

# DATA TYPES:

## NUMERIC:

* numb: signed integer range between -35,256 to 35,257 SIZE = 4bytes
* deci: datatype used to represent the floating-point number with
* precision of 4 decimal digits SIZE = 4bytes
* double: datatype used to represent the floating-point number with precision of 6 decimal digits SIZE = 8bytes
* bool: data type that can store true or false

## STRING:

* cchar: datatype used to store single character size = 1byte
* schar: datatype used to store sequence of characters SIZE = 20bytes

# VARIABLE DECLARATION:

* Variable should start with Capital letter only and not with small letter and numbers.
* eg: Xyzzy, XYZ, X1
* To declare an integer variable, we use numb keyword followed by name of the variable and can be initialized at the time of declaration. the value of the variable must be enclosed by $.
* Eg: numb Ab=12$
* For declaring a character variable, we use cchar keyword followed by name of the variable and the value should be entered in quotation marks
* Eg: cchar Ab='S’$
* For declaring a string variable, we use schar keyword followed by name of the variable and the value should be entered in inverted commas
* Eg: schar Ab=“COMPILER”.
* Values of type bool can contain only true or false.
* Declaration and all other statements are must be terminated using dollar symbol.
* We cannot use any special character at the beginning of a variable declaration

# Decision making statements:

1. if statement:

if statement is the simplest form of decision-making statement. It takes an expression and checks if the expression evaluates to True then the block of code in if statement will be executed.

eg:

numb A=20, B=20$

when(A==B):

\_print("A and B are equal")$

print("If block ended")$

## 2. if else statements:

if the expression is true then the if block is executed, otherwise the else block will be executed

-->If the if statement is indented with underscore, use the same number of underscore for else also.

-->underscores are used to denote a block of code

eg:

when (X>Y):

\_print(X)$

else:

\_print(Y)$

## 3. nested statements:

A nested statement is an if else statement placed inside another if statement.

eg:

X=15$

when (X>=0):

\_when (X==0):

\_print("Zero")$

\_else:

\_print("positive number)$

else:

\_print("Negative number")$

# Iterative statements:

Iteration statements or loop statements allow us to execute a block of statements as long as the condition is true.

1. While Loop in Suzuka :

a while loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in the program is executed.

eg:

numb Count=0$

until (Count<3):

\_Count=Count+1$

\_print("Hello world")$

else statement with while loops :

The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won’t be executed.

eg:

Count=0.

until (Count<3):

\_Count=Count+1.

\_print("Hello world")$

else:

\_print("In else block")$

## 2.Iterate loop in Suzuka:

Iterate loops are used for sequential traversal. It can be used to iterate over a range and iterators.

eg:

numb N=4,I$

iterate I to N:

\_print(I)$

## 3.Nested for loop in Suzuka:

Suzuka programming language allows to use one loop inside another loop.

eg:

numb I,N=4$

iterate I to N:

\_iterate J to I:

\_iterate k to I:

\_print(k)$;

\_print(I)$;

\_print()$;

## 4.Pass statements:

We use pass statement to write empty loops. Pass is also used for empty control statements.

eg:

numb A,B$

A=10$

B=20$

when(A<B):

\_pass$;

else:

\_print("B<A")$;

# CFG (Context Free Grammar)

## VARIABLE DECLARATION:

* Var\_dec -> <Type><Var\_list>$
* Type -> numb|deci|bool|double|schar
* Var\_list -> id<V>
* V -> ,<Var\_list>|~

## FIRST:

First(Var\_desc) = { numb, deci, bool,double,schar ,cchar}

First(Type) = { numb, deci, bool,double,schar,cchar}

First(Var\_list) = { id}

First(Var\_desc) = { ,, ~ }

## FOLLOW:

Follow(Var\_desc) = { ^ }

Follow(Type) = { Var\_desc, $ }

Follow(Var\_list) = {$}

Follow(V) = { $}

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **^** | $ | numb | deci | cchar | schar | bool | , | id |  |
| Var\_dec |  |  | Var\_dec -> <Type><Var\_list>$ | Var\_dec -> <Type><Var\_list>$ | Var\_dec -> <Type><Var\_list>$ | Var\_dec -> <Type><Var\_list>$ | Var\_dec -> <Type><Var\_list>$ |  |  |  |
| Type |  |  | Type -> numb | Type -> numb | Type -> cchar | Type -> schar | Type -> bool |  |  |  |
| Var\_list |  |  |  |  |  |  |  |  | Var\_list -> id<V> |  |
| V |  | V -> ~ |  |  |  |  |  | V ->,<var\_list> |  |  |

## VARIABLE INITIALIZATION:

* Var\_dec -> <Type><Var\_list><I>$
* Type -> numb|deci|bool|double|char|schar
* Var\_list -> id<V>
* V -> ,<Var\_list>|~
* I -> =<Init>
* Init -> Integers|Decimals|Bool|String
* Integers -> int
* Decimals -> num.num
* String -> id

## LL(1) Parser for variable initialization:

* Var\_dec -> <Type><Var\_list><I>$
* Type -> numb|deci|bool|double|char|schar
* Var\_list -> id<V>
* V -> ,<Var\_list>|~
* I -> =<Init>
* Init -> Integers|Decimals|Bool|String
* Integers -> int
* Decimals -> int.int
* String -> id

## FIRST:

First(Var\_dec) = { numb, deci, bool,double,schar ,cchar}

First(Type) = { numb, deci, bool,double,schar,cchar}

First(Var\_list) = { id}

First(V) = { ,, ~ }

First(I)={=}

First(Init)={int,int.int,0,1,”}

First(Integers)={int}

First(Bool)={0,1}

First(Decimals)={int}

First(String)={“}

**FOLLOW:**

Follow(Var\_dec) = { ^ }

Follow(Type) = { id}

Follow(Var\_list) = {=}

Follow(V) = { =}

Follow(I) = { $}

Follow(Init) = { $}

Follow(Integers) = { $}

Follow(Decimals)={$}

Follow(Strings)={$}

Follow(Bool)={$}

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **^** | $ | numb | deci | cchar | schar | bool | , | id | = | “ | 0 | 1 | int | int.int | ” |
| Var\_dec |  |  | Var\_dec> <Type><Var\_list>  <l>$ | Var\_dec> <Type><Var\_list>  <l>$ | Var\_dec> <Type><Var\_list>  <l>$ | Var\_dec> <Type><Var\_list>  <l>$ | Var\_dec> <Type><Var\_list>  <l>$ |  |  |  |  |  |  |  |  |  |
| Type |  |  | **Type -> numb** | **Type -> deci** | **Type -> cchar** | **Type -> schar** | **Type -> bool** |  |  |  |  |  |  |  |  |  |
| Var\_list |  |  |  |  |  |  |  |  | **Var\_list -> id<V>** |  |  |  |  |  |  |  |
| V |  |  |  |  |  |  |  | **V -> ,<Var\_list>** |  | **V -> ~** |  |  |  |  |  |  |
| I |  |  |  |  |  |  |  |  |  | **I -> =<Init>** |  |  | < |  |  |  |
| Init |  |  |  |  |  |  |  |  |  |  | **Init -> String** | **Init -> Bool** | **Init -> Bool** | **Init -> Integers** | **Init -> Decimals** |  |
| Integers |  |  |  |  |  |  |  |  |  |  |  |  |  | **Integers -> int** |  |  |
| Decimals |  |  |  |  |  |  |  |  |  |  |  |  |  |  | **Decimals -> int.int** |  |
| Strings |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Bool |  |  |  |  |  |  |  |  |  |  |  | **Init -> Bool** | **Init -> Bool** |  |  |  |

**1)WHEN:**

C -> <W><E>

W -> when(< E>)<T\_cln>\_<P>

P -> print(V)$|Expression

V -> “id”|id|~

T\_cln -> :

Expression -> id<M>

M -> <Op><Expression>|~

Op -> +|-|\*|/|%|<|>|<=|>=

E -> else<T\_cln>\_<P>|~

**2)UNTIL:**

until (Count<3):

C -> <U>

U -> Until(<E>)<T\_cln>\_<P>

P -> print(V)$|Expression

V -> “id”|id|~

T\_cln -> :

Expression -> id<M>

M -> <Op><Expression>|~

Op -> +|-|\*|/|%|<|>|<=|>=

**3.Iterate**

iterate I to N:

loop -> iterate <id> to <id> <T\_cln>\_<P>

P -> print(V)$|Expression

V -> “id”|id|~

T\_cln -> :

Expression -> id<M>

M -> <Op><Expression>|~

Op -> +|-|\*|/|%|<|>|<=|>=

Parse tree example for print:

print

D L “ Text “

XYZ

Parse tree example for When:

When

when Relation T-ch

L-parn Expre Relati Expre R-para :

)

j

<

I

(

**LEXICAL ANALYSIS:**

digit [0-9]

id [A-Z][a-zA-Z0-9]\*

unsigned\_integer {digit}\*

signed\_integer -{digit}\*

unsigned\_float {digit}\*\.{digit}\*[e[+|-]?{digit}\*]?

keyword when|until|iterate|while|return|pass|else

arop [+\-\\*\/\%]

p print

%{

#include<stdio.h>

#include<string.h>

int cnt=0,i=0,j=0;

char st[20][10];

int c=0,m=0;

int look\_up(char st[10][10],char \*id,int n);

%}

%%

"\n" {c=0;}

"@" {c=1;}

"/@" {m=1;}

"@/" {m=0;}

{keyword} {if(!c&&!m)fprintf(yyout,"\n%s is Keyword",yytext);}

{unsigned\_integer} {if(!c&&!m)fprintf(yyout,"\n%s is Unsigned Integer",yytext);}

{signed\_integer} {if(!c&&!m)fprintf(yyout,"\n%s is Signed Integer",yytext);}

{unsigned\_float} {if(!c&&!m)fprintf(yyout,"%s is Unsigned Float.",yytext);}

{id} {if(c==0&&m==0){

if (!lookup(st,yytext,i)){

strcpy(st[i++],yytext);

fprintf(yyout,"\n%s is Identifier",yytext);

cnt++; }}}

"<=" {if(!c&&!m)fprintf(yyout,"\n %s is Relational operator Lessthan or Equal to",yytext);}

"<" {if(!c&&!m)fprintf(yyout,"\n %s is Relational operator Lessthan",yytext);}

">=" {if(!c&&!m)fprintf(yyout,"\n %s is Relational operator Greaterthan or Equal to",yytext);}

">" {if(!c&&!m)fprintf(yyout,"\n %s is Relational operator Greaterthan",yytext);}

"==" {if(!c&&!m)fprintf(yyout,"\n %s is Relational operator Equal to",yytext);}

"!=" {if(!c&&!m)fprintf(yyout,"\n %s is Relational operator Not Equal to",yytext);}

"=" {if(!c&&!m)fprintf(yyout,"\n %s is Assignment operator",yytext);}

{arop} {if(!c&&!m)fprintf(yyout,"\n %s is arithmetic operator",yytext);}

"$" {if(!c&&!m)fprintf(yyout,"\n %s is statement terminator",yytext);}

{p} {if(!c&&!m)fprintf(yyout,"\n %s is function",yytext);}

. {if(!c&&!m)fprintf(yyout,"\n %s is delimiter",yytext);}

%%

int main(int argc,char\*\*argv)

{

yyin=fopen(argv[1],"r");

yyout=fopen(argv[2],"w");

yylex();

printf(" No. of identifiers are : %d ",cnt);

printf("\n the contents of symbol table are :\n");

for(j=0;j<i;j++)

printf("\n %s",st[j]);

return 0;

}

int yywrap()

{

return 1;

}

int lookup(char st[10][10],char \*id,int n)

{

for(j=0;j<n;j++)

if(!strcmp(st[j],id))

return 1;

return 0;

}

**SOURCE FILE:**

numb A=10$

numb B=20$

numb C=A+B$ @ sum of two numbers

when(A<B): @ conditional construct

\_print("B")$

else:

/@ SUZUKA COMPILER @/

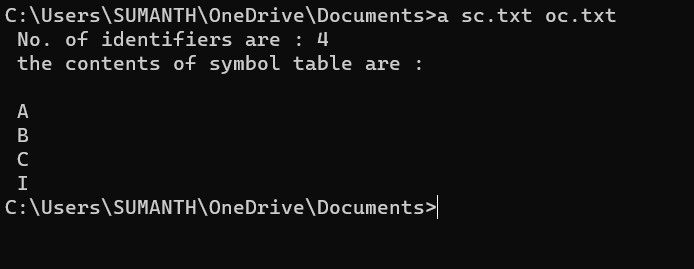
\_print("A")$

numb I =0$

iterate I to C: @ Iteration

\_print(I)$

**OUTPUT:**



**OUTPUT FILE:**

numb is Keyword

is delimiter

A is Identifier

= is Assignment operator

10 is Unsigned Integer

$ is statement terminator

numb is Keyword

is delimiter

B is Identifier

= is Assignment operator

20 is Unsigned Integer

$ is statement terminator

numb is Keyword

is delimiter

C is Identifier

= is Assignment operator

+ is arithmetic operator

$ is statement terminator

is delimiter

when is Keyword

( is delimiter

< is Relational operator Lessthan

) is delimiter

: is delimiter

is delimiter

is delimiter

\_ is delimiter

print is function

( is delimiter

" is delimiter

" is delimiter

) is delimiter

$ is statement terminator

else is Keyword

: is delimiter

\_ is delimiter

print is function

( is delimiter

" is delimiter

" is delimiter

) is delimiter

$ is statement terminator

numb is Keyword

is delimiter

I is Identifier

is delimiter

= is Assignment operator

0 is Unsigned Integer

$ is statement terminator

iterate is Keyword

is delimiter

is delimiter

t is delimiter

o is delimiter

is delimiter

: is delimiter

is delimiter

\_ is delimiter

print is function

( is delimiter

) is delimiter

$ is statement terminator