

Dungeons and Dragons Project Final Writeup

This project is an automation of the tedious task of creating a Dungeons & Dragons character. There are many calculations involved and creating a character by hand can take about 8 hours. So we decided to use ELI to create a language which would allow you to input the base character data and using a file with some basic race data, would output that character's specifications at every level of play and at leveling completion. Using this program should reduce the amount of time to create a character by up to 3 hours.

This paper has three purposes: to describe the language itself, the output and a specification of the program.

The Language

The language includes two or more files: the character's data and the data describing that character's class (a .class file). The class file is basic data about a specific class of character that shouldn't need to be changed from program to program. Below is an example file:

```
Fighter :
  HD = 10 /*basic data section*/
  Skills = 2
  BAB = 4
  Saves :
    FORT
  end;
  SpecialAbilities :
    Level 1 :
      Evasion :
        +2 TO REF
        /*PHP p.92*/
      end;
    end;
    Level 2 :
      BONUS FEAT end;
    end;
    Level 4 :
      BONUS FEAT end;
    end;
    Level 6 :
      BONUS FEAT end;
    end;
    Level 8 :
      BONUS FEAT end;
    end;
    Level 10 :
      BONUS FEAT end;
```

```

end;
Level 12 :
    BONUS FEAT end;
end;
Level 14 :
    BONUS FEAT end;
end;
Level 16 :
    BONUS FEAT end;
end;
Level 18 :
    BONUS FEAT end;
end;
Level 20 :
    BONUS FEAT end;
end;
end;
ClassSkills :
    Climb
    Craft
    Handle
    Animal
    Intimidate
    Jump
    Hide
    Swim
end;
end;

```

The file starts creating a block that starts with the type of class that this file will describe *Fighter* and ends with an *end;*. There are then four sections inside this: basic data, *Saves*, *SpecialAbilities*, and *ClassSkills*. The sections begin with the title then colon and ends with an *end;* except for the basic data section which has all of its parts in a list.

The basic data section specifies the amounts of each of the following: *HD*, *Skills*, and *BAB*. *HD* means hit dice and it specifies the type of die should be used when calculating hit points, in this case a 10 sided die. *Skills* is the number of skill points that character will receive each level. *BAB* or the base attack bonus specifies how many increases that character's to hit value gets for every 4 levels. It has a max value of 4. The *Saves* section specifies the "good saves" or saves that will increase at a higher level than the regular saves. The *SpecialAbilities* section specifies what feats that the character will receive at each level. One option for each level is to specify the feat itself (in this case *Evasion*) which includes an optional modifier to a skill +2 *TO REF* and a C style comment */*PHP p.92*/*. The other option is to say *BONUS FEAT* which will allow that class an additional feat from the Feats section of the other input file in addition to the number it would normally get. The last section is the *ClassSkills* section which specifies the base skills of that class. These skills mean that that skill will increase at a greater rate than normal skills.

The main input file of the program is the file that specifies that specific character's data, below is an example:

Group DragonFist :

```
Group Data :
  Feat Progression = 2
  Level = 10
end;

Character Pip :
  Race = Halfling
  include Fighter.class ;
  Weapons :
    Melee = #Dagger Dtype 4
    Ranged = #Crossbow Dtype 8
end;
Stats :
  STR = 12
  DEX = 18
  CON = 14
  INT = 10
  WIS = 18
  CHA = 12
  Primary Stat = DEX
end;
Skills :
  Hide ( DEX ) = 4
  MoveSilently ( DEX ) = 4
end;
Feats :
  Stealth :
    +2 TO Hide
    +2 TO MoveSilently
    /*page 104 PHB*/
  end;
  WeaponFocus :
    +1 TO #Dagger
    /*page 105 PHB*/
  end;
  PointBlankShot :
    +1 TO #Crossbow
    +1 TODAMAGE #Crossbow
    /*page 104 PHB*/
  end;
  Dodge :
    +1 TO AC
    /*page 101 PHB*/
  end;
  IronWill :
    +2 TO WILL
    /*page 102 PHB*/
  end;
  ImprovedInit :
    +4 TO INIT
    /*page 102 PHB*/
  end;
end;
```

```

    MagicItems :
      GlovesOfDex :
        +4 TO DEX
        /*page 317 DMG*/
      end;
      DaggerOFVenom :
        +2 TO #Dagger
        +2 TODAMAGE #Dagger
        /*once per day this dagger will coat itself with
        Scorpion Poison
        page 297 DMG*/
      end;
    end;
  end;
end;

```

This file allows you to specify multiple characters and their data in a group. You start by specifying the group and its name *Group DragonFist* and end with an *end;*. There are then two sections: *Group Data* and *Character*. *Group Data* ends with an *end;* and specifies the feat progression and the level at which the group is to be calculated to. *Feat Progression* specifies at what rate feats are to be acquired, in this case every other level.

The main section of this file is the Character section of which there can be multiples of. It starts by specifying the character's name *Character Pip* : and ends with an *end;*. There are six sections in Character: base information, *Weapons*, *Stats*, *Skills*, *Feats*, and *Magic Items*. They all begin with the title then colon and end with an *end;*. The base information section has two lines. *Race* which specifies the race of the character and *include* which specifies the class input file to use and ends with a ;.

The *Weapons* section specifies the *Melee* and *Ranged* weapons that this character will have. The weapon name has to begin with a # and the *Dtype* 4(or other number) specifies what sided die should be used to determine what damage is done with that weapon.

The *Stats* section specifies the level of the basic character stats (*STR*, *DEX*, *CON*, *INT*, *WIS*, *CHA*) and the *Primary Stat*. The basic stats, specified with an = and the number, are the basic statistics of that character and are used in almost every other calculation in the program. The *Primary Stat* specifies what stat will be increased every four levels automatically.

The *Skills* section specifies the skills that this character has and what stat that they modify. After the name of the skill *Hide* and inside the parentheses there is the stat that will be modified by the skill (*DEX*). The number after the equals specifies when that skill is to be added too, in the case of 4 it will be added to every 4th level. If that skill is in the *ClassSkills* section of the class file, the stat will be incremented by one otherwise it is incremented by a half. The half value doesn't mean anything to the stat until there are two of them to add together and make one.

The *Feats* section specifies the feat pool from which this character can draw from each level if the *Feat Progression* says it can or *BONUS FEAT* is specified in the current level in the class file. The feats are specified by giving the feat name *Stealth*, a colon and then an *end;* at the end. A feat can hold a few different things. It has to have a c-style comment */*page 104 PHB*/* and at least one modifier. The modifier begins with a +2(or other number) then a TO and then one of a few different values: skills *Hide*(examples in italics), weapons *#Dagger*(note that once again the weapon is specified with a # at the

start), weapon damage *TODAMAGE #Dagger*(note that there is no space between the *TO* and the *DAMAGE*), saves *WILL*, stats *DEX*, armor class *AC*, and initiative *INIT*.

The *MagicItems* section specifies the magic items that that character has in its possession. These are only used in the final calculations and not in the level calculations. They begin with the name and a colon and end with an *end;*. The insides are the same as a feat: a modifier or modifiers and a c-style comment.

Output

The output has all the data about the group and all characters in it up to the level specified. It starts with the group name and then each character in the group has three sections: title and stat information, level by level details, and the final look.

Group: DragonFist

Name: Pip Race: Halfling

Stats:

STR: 12

DEX: 18

CON: 14

INT: 10

WIS: 18

CHA: 12

Level 1:

+1 BAB

+2 FORT

Feat(s): Stealth, WeaponFocus

Skill(s): +2 Hide, +2 MoveSilently

Level 2:

+1 BAB

+1 FORT

Feat(s): PointBlankShot

Skill(s): +0.5 Hide, +0.5 MoveSilently

Final Look:

Name: Pip Race: Halfling

Medium-size Humanoid

Fighter 2

CR: 2

HD: 1d10+14

hp:19

Initiative: +6

Speed 30ft.

AC: 16

Attacks:

#Dagger +6 Melee (1d4+3)

#Crossbow +9 Ranged (1d8+1)

Saves:
FORT: 5
REF: 6
WILL: 4
Stats:
STR: 12 | 1
DEX: 22 | 6
CON: 14 | 2
INT: 10 | 0
WIS: 18 | 4
CHA: 12 | 1

Skills:
Hide(DEX): +10
MoveSilently(DEX): +10
Feat(s):

Stealth /*page 104 PHB*/
WeaponFocus /*page 105 PHB*/
PointBlankShot /*page 104 PHB*/

Special Abilities:

Magic Items:

GlovesOfDex /*page 317 DMG*/
DaggerOFVenom /*once per day this dagger will coat itself with
Scorpion Poison
page 297 DMG*/

The title and stat information just give the name, race, and starting stats. The level field has changes done in that particular level including: changes to the saves, feat additions, and skill changes. The Final Look includes all final information about the character. This includes the final values of: all character statistics like name and race, the level of class they have reached *Fighter 2*, challenge rating *CR: 2*, hit dice *HD: 1d10+14*, hit points *hp: 19*, initiative *Initiative: +6*, speed *Speed 30ft.*, armor class *AC: 16*, attack data (current values of the weapons) *#Dagger +6 Melee (1d4+3)*, saves *FORT: 5*, statistics and the values they increased by *DEX: 22 / 6*, skills *Hide(DEX): +10*, and a list of the feats and special abilities that were used on this character.

Specification

In this part of our project we had to deal with the following issues: Iteration, Strict Dependencies Between Calculations, Name Analysis, Error Checking, Making Keys Available, and Using a Second Input File.

Iteration

We realized early on that we would not only have to do certain computations in the rules but that those same computations would need to be done for each level the character goes through. To do this we used something called ITERATE:

RULE : Iterator ::= X

COMPUTE

```

X.cnt = 1;
Iterator.done = UNTIL GT (X.cnt, INCLUDING Group.Level)
    ITERATE X.cnt = X.incr;

```

What this does is iterate any computations that are dependant on the variables in the computation until the statement before the ITERATE is true, doing whatever is after the ITERATE every iteration. In this case, we iterate all computations that depend on X.cnt until the “UNTIL GT (X.cnt, INCLUDING Group.Level)” is true while doing “X.cnt = X.incr” every iteration. This means that until X.cnt is greater than the level you want to go to, iterate X.cnt's computations (one of which is to add 1 to X.incr) and to set X.cnt to X.incr each iteration.

To get the computations that we needed in the iteration, we added the following rule:

```

                                END;
RULE : X      ::= Character      COMPUTE
    CHAINSTART HEAD.PRO = X.cnt;
    X.incr = ADD (X.cnt, 1)<-TAIL.PRO;
                                END;

```

So now we are running through the chain PRO then adding one to X.incr every iteration.

The computations in the chain have to do with leveling up such as: skill increases, feat computations, special ability computations, weapon hit score increases, and stat increases.

Strict Dependencies Between Calculations

Every single calculation can be categorized in one of three ways: calculations that need to happen before the iteration begins, calculations that need to happen during the iteration, and calculations that need to happen after the iteration is completed. There are very strict dependencies between these categories because otherwise Eli might decide to do a calculation during the iteration that shouldn't be there and such calculations would run multiple times and would produce incorrect results at the end of the iteration.

The calculations that happen before the iterations start are set up calculations for the iteration so that the iteration process can run correctly with the correct values. Below is an example of the code that tests whether the same name was used for the ranged and melee weapons.

```

RULE : Melee      ::= 'Melee' '=' WeaponDef 'Dtype' Number  COMPUTE
    Melee.MKey = WeaponDef.Key<-INCLUDING Character.PreCalc;
    Melee.NameTest = SetUsed(Melee.MKey, 1, 2);
    Melee.a = IF (EQ (GetUsed(Melee.MKey,0), 2),
    message (ERROR,
        "cannot use same weapon name for both melee and Ranged",
        0, COORDREF))<- Melee.NameTest;
    /* can only use weapon name once */
    Melee.PreItr = ORDER( ResetTotal(Melee.MKey, 0),
    ResetDType(Melee.MKey, Number),
    ResetDTotal(Melee.MKey, 0),
    ResetPStat(Melee.MKey, KeyInEnv( INCLUDING Iterator.Env,

```

```
MakeName("STR"))))<-Melee.a;
```

In this example, the three lines: assign to `Melee.a`, assign to `Melee.NameTest`, and the message are for testing whether the same name was used for the melee and ranged weapons. This works by seeing if there are any previous occurrences of the name and sending an error if there isn't. If this was to occur during every iteration, it would send an error out every time because the name occurred when it was declared. It would be detecting that initial declaration and not a duplicate every time. We fixed this by making the `ORDER` statement depend on the completion of `Melee.a` and also assign to `Melee.PreItr`, which is setup to be completed before the iterations begin.

The calculations that need to be run during the iteration are those that need to occur every time the character levels up. These need to be specifically attached to the chain `PRO` (Process Right Order), which controls what is done during the iteration. Not only do they have to be attached to the chain, but they have to be attached in a specific order due to the fact that some of the calculations during the level-up can be effected by other calculations during the level-up. One example is the `BONUS FEAT` ability associated with the character's class (in the class file):

```
Level 2 :  
    BONUS FEAT end;  
end;
```

This node will grant the character an extra feat at this level. In order to ensure that the character gets this extra feat at this level, the special abilities for this character need to be processed before the character's feat. This example demonstrates what happens when the feat is calculated before the special abilities.

```
(in RULE : Feat    ::= FeatName ':' Effects Comments 'end;'    COMPUTE)  
.SKey = KeyInEnv( INCLUDING Iterator.Env, MakeName("FEAT"))<-INCLUDING  
    Character.PreCalc;  
Effects.run = FFirstRun(.Key,.SKey)<-Feat.PRO;  
  
(in RULE : Ability ::= 'BONUS FEAT' 'end;'    COMPUTE)  
Ability.PRO = IF (.cond, ResetTotal(KeyInEnv( INCLUDING Iterator.Env,  
    MakeName("FEAT")), ADD(GetTotal(KeyInEnv( INCLUDING Iterator.Env,  
    MakeName("FEAT")),0), 1)));
```

In the `Ability` rule, the value stored at the `FEAT` key holds how many feats can to be assigned in that particular level. So because this is a `BONUS FEAT` rule, it adds one to the value at the `FEAT` key. Then in the `Feat` rule, feats are used only if the value at the `FEAT` key is greater than zero. If it is greater than zero, the feat is used and then in `FFirstRun` the value at the `FEAT` key is decremented by one. So if the calculations in the `BONUS FEAT` rule are not done before those in the `Feat` rule, the value stored at the `FEAT` key will be one smaller than it should be which will cause there to be one less feats being used. The `Ability` rule being completed before the `Feat` rule is guaranteed by the chain which can't continue on until all of the feats are computed.

The Calculations that need to be processed after the iteration is finished involve the final output or modifying values that could affect the calculations done during the iteration that shouldn't be calculated until the end. The magic items for this character

need to be processed at the end because it is possible for the magic item to affect the INT value which can affect the number of skill points. The following is an example.

```
(in RULE : Character ::= 'Character' CharName ':' CharDetails Weapons Stats
    SkillAllot FeatList Magic 'end;' COMPUTE)
ResetTotal( KeyInEnv( INCLUDING Iterator.Env, MakeName("SKILLS")),
ADD(GetTotal(KeyInEnv( INCLUDING Iterator.Env, MakeName("SKILLS")),0),
ADD(GetRate(KeyInEnv( INCLUDING Iterator.Env, MakeName("SKILLS")),2),
STATMod(GetTotal(KeyInEnv( INCLUDING Iterator.Env,
MakeName("INT"),8))))))<-Character.PRO;

(in RULE : MagicItem ::= ItemName ':' Effects Comments 'end;' COMPUTE)
Effects.run = 1<-INCLUDING Iterator.done;
```

In the Character rule, the skill points are being modified by the INT value. It is possible for a magic item to add to the INT value. So we want to make sure that the skill points are computed before the magic items. In rule MagicItem, magic items are being computed. Effects.run controls whether the effects (insides) of that particular magic item are applied to the character. It is set only if Iterator.done is done which means that the iteration is completed. Skill points are computed in the iteration, so magic items are now computed after the skill points.

Name Analysis

Our program uses Name Analysis for the following variables: Skill, Stat, Save, and Weapon. These can be declared and used later on in the program. See the examples below.

The Skill definition is in the character creation file in the Skills section when you need to specify what skills each character has and their values. Skill use is used in the Feats or MagicItems section when you specify a feat and what effect it has on the skills.

The Stat definition is in the Stats section of the character which is where you define what values each of the 6 stats have. Stat use is done when you declare a skill, you have to associate a stat with that skill because the final skill modifier depends on the stat. Stat use is also used when a feat or magic item needs to modify a stat.

The Save definition is done in the second input file (.class) when you need to specify what saves are associated with that type of character. Save use occurs in the first input file when a feat or magic item modifies that save.

The Weapon definition is under the Weapons heading when you need to define the melee and ranged weapons that the character has. The weapon use occurs in the Feats or MagicItems section when a feat or magic item modifies a weapon's statistics.

```
Weapons :
    Melee = Dagger weapon definition
    Ranged = Crossbow
end;
```

```
Stats :
    STR = 12 stat definition
```

```
DEX = 18
CON = 14
INT = 10
WIS = 8
CHA = 12
Primary Stat = DEX
end;
```

```
Skills :
  Hide (STR) = 4 skill definition and (stat use)
  MoveSilently (CON) = 4
end;
```

```
Feats :
  Stealth :
    +2 TO Hide skill use
    +2 TO MoveSilently skill use
    /*page 104 PHB*/
  end;
  WeaponFocus :
    +1 TO Dagger weapon use
    +2 TODAMAGE Crossbow
    /*page 105 PHB*/
  end;
  GreatFortitude:
    +2 TO FORT save use
    /*page 102 PHB*/
  end;
end;
```

```
(In second input file)
Saves :
  FORT save definition
end;
```

This name analysis is done with symbol computations that inherit either IdDefScope or IdUseEnv. Here is an example of Skill:

```
SYMBOL SkillDef INHERITS IdDefScope COMPUTE
  SYNT.Sym = TERM;
END;
SYMBOL SkillUse INHERITS IdUseEnv COMPUTE
  SYNT.Sym = TERM;
END;
```

Then in the appropriate places and rules in the lido file we use either SkillDef or SkillUse. Example:

```

RULE : Skill      ::= SkillDef '(' StatUse ')' '=' Number      etc...
RULE : EffectKind ::= SkillUse                                COMPUTE
      EffectKind.Key = SkillUse.Key
                                                                END;

```

Making Keys Available

One problem we ran into was that we needed some of the variables and their keys available whether the user declares them or not. An example of this are the saves. They are declared in the Saves section of the second input file but are needed for computations later and in the final output. We have to make sure that each of the saves are represented in the def table whether the user declared them or not. To do this we use the following line for FORT to put it in the def table:

```
ResetTotal(DefineIdn( INCLUDING Iterator.Env, MakeName("FORT")), 0);
```

This is done as one of the first computations in the tree. Other than the saves, the other keys that are needed are: INIT, AC, SKILLS, FEAT, and the stats.

Error Checking

We wanted to place some restrictions on what the user was putting in their input file and throw errors if they didn't do the right thing. We had a few cases of this: we couldn't open the specified second input file, the melee and ranged weapon are the same, the same stat was declared more than once, and the same skill was defined more than once.

As an example, if the melee and ranged weapons are the same our program will throw an error and output. Here would be the code for this in the Melee rule (it has a counterpart in the Ranged rule):

```

Melee.NameTest = SetUsed(WeaponDef.Key, 1, 2);
IF (EQ (GetUsed(WeaponDef.Key,0), 2),
     message (ERROR,
              "cannot use same weapon name for both melee and Ranged",
              0, COORDREF))<- Melee.NameTest;

```

This works by setting the Used attribute to 1 if that particular weapon has not been defined before and 2 if it has. If Used is 2, throw the error.

Using a Second Input File

Another problem we ran into was that we needed a second input file of character data. We didn't want the user to have to put this data in the first input file because it is a lot of data about specific types of characters that doesn't change. To do this, we created a separate section of our lido file to create an AST for it and a couple rules in the first AST to open the file and insert the second AST into the first.

We decided that the best place in the first AST to do this was create a rule called

FileInclusion which looked for a “include something.class” in the input file and then opened that file and inserted it's data in the first input file. FileInclusion came from the rule FileInclusion1 which as parameters has FileInclusion and ClassStats which is the top node of the second input file. Here are the two rules:

```
RULE : FileInclusion1 ::= FileInclusion ClassStats      END;
RULE : FileInclusion ::= 'include' FileName           COMPUTE
      .InpFileEx = NewInput(StringTable (FileName)) BOTTOMUP;

IF (NOT (.InpFileEx),
    message (ERROR, CatStrInd ("can not open file ", FileName),
            0, COORDREF));                                END;
```