Universidade Federal de Santa Catarina

INE5426 – Construção de Compiladores 2016/2

Projeto 1 – Linguagem: Łukasiewicz Versões 0.1 a 1.0

Doutor Jan Łukasiewicz foi um filósofo e matemático conhecido por, entre outros fatos, pela criação da Notação Pré-Fixada, também chamada de Notação Polonesa.

Os grupos irão desenvolver um compilador para a linguagem Łukasiewicz que gera a árvore de sintaxe para um dado código de entrada e posteriormente imprime informações relevantes do código, sendo as operações escritas em Notação Prefixada.

A linguagem de programação Łukasiewicz é uma linguagem procedural, estaticamente tipada, fortemente tipada, com escopos estáticos e palavras reservadas. Ela tem múltiplas versões, cada qual com características e mecanismos adicionais. As versões são especificadas nas próximas seções deste documento.

Índice

Informações básicas do projeto 4
Versão 0.1 – Linguagem introdutória 5
Exemplo de código 5
Saída do compilador 5
Tratamento de erros 5
Versão 0.2 – Ponto flutuante e valores booleanos 7
Exemplo de código 7
Saída do compilador 7
Tratamento de erros 7
Versão 0.3 – Conversão de tipos 8
Exemplo de código 8
Saída do compilador 8
Tratamento de erros 8
Versão 0.4 – Expressões condicionais 9
Exemplo de código 9
Saída esperada 9
Tratamento de erros 10
Versão 0.5 – Laços 11
Exemplo de código 11
Saída esperada 11
Tratamento de erros 11
Versão 0.6 – Múltiplos escopos 12
Exemplo de código 12
Saída esperada 12
Tratamento de erros 12
Versão 0.7 – Funções 13
Exemplo de código 13
Saída esperada 13
Tratamento de erros 14
Versão 0.8 – Arranjos 15
Exemplo de código 15
Saída esperada 15
Tratamento de erros 15
Versão 1.0 - Ponteiros 16
Exemplo de código 16

Saída esperada 16
Tratamento de erros 16
Termos para saída do compilador e mensagens de erro 18
Versão 0.1 18
Saída padrão 18
Erros 18
Versão 0.2 18
Saída padrão 18
Erros 18
Versão 0.3 18
Saída padrão 18
Versão 0.4 18
Saída padrão 18
Erros 18
Versão 0.5 18
Saída padrão 18
Versão 0.7 19
Saída padrão 19
Erros 19
Versão 0.8 19
Saída padrão 19
Erros 19
Versão 1.0 19
Saída padrão 19
Erros 19

Informações básicas do projeto

Os grupos serão formados por uma a três pessoas. Serão distribuídos os seguintes papéis e responsabilidades entre as pessoas do grupo:

- Gerente de projeto: responsável pela documentação do projeto, respeitos às datas de entrega e comunicação com o professor.
- Arquiteta de sistema: responsável pela arquitetura do compilador.
- Testadora: responsável pelo planejamento dos testes para verificação e validação do sistema.

No caso de grupos com menos de três pessoas, membros terão mais de uma responsabilidade.

O compilador deverá consumir todo o código de entrada antes de começar a imprimir as mensagens de saída na saída padrão.

Erros deverão ser alertados o mais cedo possível através de mensagens na saída de erros.

Cada grupo irá manter seu projeto em um repositório Git acessível ao professor e integrado com seu canal privado no Slack. Mensagens de *commits* deverão ser significativas para a compreensão do código e evolução do projeto.

O repositório do grupo deverá conter uma pasta com nome ‘projeto1’, onde um arquivo Makefile será mantido para a compilação do compilador da linguagem Łukasiewicz. O arquivo gerado ao final da compilação deverá se chamar ‘lukacompiler’, o qual será usado para a realização de testes.

As palavras chave da linguagem serão identificadas nas próximas seções através de texto sublinhado.

Versão 0.1 – Linguagem introdutória

A versão introdutória da linguagem tem as seguintes características:

I. Um tipo inteiro int com valores representados por números naturais compostos por dígitos decimais.
II. Quatro operadores binários: soma +, subtração -, multiplicação * e divisão /.
III. Um operador unário -.
IV. Parênteses ().
V. Variáveis com nomes começados por um caractere (‘a’ a ‘z’, maiúsculo ou minúsculo), podendo ser seguidos por outros caracteres, dígitos decimais e ‘_’.
VI. Atribuições =.

As diferentes instruções da linguagem são separadas apenas pela marcação de fim de linha.

Múltiplas variáveis de mesmo tipo podem ser declaradas em uma mesma linha, necessitando-se apenas da separação entre seus nomes com uma vírgula ,. Cada variável pode receber uma atribuição de valor inicial de valores constantes.

A precedência entre os operadores é a mesma da matemática padrão.

Não é possível fazer múltiplas atribuições em uma mesma linha.

Exemplo de código

```
int a_
int    BB, c
int d=0,          e1=1
a_ = d+2*3
BB = (-a_)/  12-1
c = e1*e1/a_
```

Saída do compilador

```
int var: a_
int var: BB, c
int var: d = 0, e1 = 1
= a_ + d * 2 3
= BB - / -u a_ 12 1
= c * e1 / e1 a_
```

Tratamento de erros

1. No caso de erros léxicos (símbolos e palavras desconhecidas), o compilador deve apenas emitir uma mensagem de erro informando sobre o caractere ou palavra desconhecida, não passando nenhum token para a análise sintática. Exemplo:
   ```
   int a&
   [Line 1] lexical error: unknown symbol &
   int #####a
   [Line 2] lexical error: unknown symbol #####
   ```
2. No caso de um erro sintático, o compilador deve ignorar todo o bloco de código comprometido (declaração de variáveis, atribuições, etc.). A mensagem deve seguir similar a original. Exemplo:
   ```
   int 10b
   ```

```
[Line 1] syntax error
```

3. No caso de uso de uma variável ainda não declarada, uma mensagem de erro semântico deve ser impressa. Exemplo:
   ```
   A = 2
   [Line 1] semantic error: undeclared variable A
   ```
4. No caso de uma variável ser declarada novamente, a declaração original deve ser mantida e a nova declaração deve ser ignorada. Uma mensagem de erro deve ser emitida. Exemplo:
   ```
   int a, a
   [Line 1] semantic error: re-declaration of variable a
   ```

Versão 0.2 – Ponto flutuante e valores booleanos

Esta versão adiciona as seguintes características à linguagem:

I. Tipo ponto flutuante float com valores representados por números reais compostos por dígitos decimais e uma vírgula.
II. Tipo booleano bool com valores true e false.
III. Operações unárias e binárias sobre valores de ponto flutuante.
IV. Operadores relacionais para comparação entre inteiros ou valores de ponto flutuante igual ==, diferente !=, maior >, menor <, maior ou igual >= e menor ou igual <=.
V. Operadores binários booleanos e & e ou |.
VI. Operador unário booleano de negação !.

Esta versão não possibilita a conversão entre tipos. Operações relacionais geram valores booleanos.

A precedência dos operadores para valores inteiros e de ponto flutuante é a padrão. A precedência para expressões booleanas é: [maior] operador unário, operadores relacionais, operadores binários.

Exemplo de código

```
float f=1.0, g=0., h= .10, i
bool b = true
i = -f*g-h/2.1
b = ! (i > 0.0) | (i < -2.3)
```

Saída do compilador

```
float var: f = 1.0, g = 0., h = .10, i
bool var: b = true
= i - * -u f g / h 2.1
= b | ! > i 0.0 < i –u 2.3
```

Tratamento de erros

5. No caso de tipos incompatíveis usados para uma operação, uma mensagem de erro semântico deve ser emitida. No caso de operações aritméticas ou atribuições, o tipo esperado é o tipo do operando à esquerda. Para reduzir o encadeamento de erros, o tipo retornado por essas operações deve ser o tipo do operando à esquerda. Operações relacionais e booleanas retornam sempre booleanos. Em caso de erro, o menos unário retorna o tipo inteiro. Exemplo:

```
int a = 1.0
[Line 1] semantic error: attribution operation expected integer but received float
a = a + true
[Line 2] semantic error: addition operation expected integer but received boolean
```

Versão 0.3 – Conversão de tipos

Esta versão adiciona as seguintes características à linguagem:

I. Coerção de valores int para valores float em operações binárias com outros valores float.
II. *Casting* entre quaisquer tipos básicos através das operações unárias [int], [float] e [bool].

As operações de *casting* tem a menor precedência possível.

Exemplo de código

```
int i = 0, j
float f = 1.1
bool b = true
j = [int] [int] i + f
i = [int] j
b = b & [bool] f
f = ([float] b) + 0.0
```

Saída do compilador

```
int var: i = 0, j
float var: f = 1.1
bool var: b = true
= j [int] [int] + [float] i f
= i [int] j
= b & b [bool] f
= f + [float] b 0.0
```

Tratamento de erros

Não há nada de novo de tratamento de erros.

Versão 0.4 – Expressões condicionais

Esta versão da linguagem inclui estruturas de execução condicional if-then e if-then-else. O teste da expressão condicional recebe um valor ou variável booleana.

Chaves {} são usadas para marcar o início da região then. No caso da presença da cláusula else, ela deve vir na mesma linha que o fechamento de chaves do then. Nada mais deve constar nessas linhas.

Na saída padrão, o corpo de cada then e else altera a tabulação em dois espaços em branco cumulativos.

Os corpos das expressões condicionais podem ser vazios e podem incluir a declaração de novas variáveis.

Exemplo de código

```
int a = 0, b = 1, c, d
  bool teste_falso = false
if a > b
then {
      if (a > 0)
then {
c = 10
}

}

if teste_falso
then {
d = 0

} else {
            d = 20
}
```

Saída esperada

```
int var: a = 0, b = 1, c, d
bool var: teste_falso = false
if: > a b
then:
  if: > a 0
  then:
    = c 10
if: teste_falso
then:
  = d 0
else:
  = d 20
```

Tratamento de erros

6. Testes devem ser valores booleanos. No caso de um tipo diferente no resultado do teste, um erro semântico deve ser gerado. Exemplo:

```
int a = 0
if a
[Line 2] semantic error: test operation expected boolean but
received integer
```

Versão 0.5 – Laços

Esta versão da linguagem inclui laços for em um estilo similar à linguagem C.

Os laços seguem a estrutura for inicialização, teste, iteração. O trecho de teste é obrigatório, enquanto os outros trechos são opcionais. A inicialização e a iteração podem conter apenas uma atribuição. O teste pode conter apenas um teste. O corpo do laço é definido por parênteses, pode ser vazio e pode conter a declaração de novas variáveis.

Na saída padrão, o corpo de cada laço altera a tabulação em dois espaços em branco cumulativos.

Exemplo de código

```
int i, j = 0
for , j < 10 , j = j + 2 {
}
for i = 0, i < 10   , i = i + 1 {
int temp
temp = j + i
j = temp
}
j = j + 0
```

Saída esperada

```
int var: i, j = 0
for: , < j 10, = j + j 2
do:
for: = i 0, < i 10, = i + i 1
do:
  int var: temp
  = temp + j i
  = j temp
= j + j 0
```

Tratamento de erros

7. Os testes dos laços devem respeitar às mesmas regras que os testes de expressões condicionais e devem gerar os mesmos erros.

Versão 0.6 – Múltiplos escopos

Esta versão da linguagem adiciona a diferenciação entre escopos.

A linguagem trabalha com um escopo global e um novo escopo interno para cada corpo de then, else e for. Dentro de novos escopos, novas variáveis com o mesmo nome de variáveis de escopos externos podem ser declaradas. O tempo de vida das variáveis fica também atrelado à duração do escopo onde elas são declaradas.

Exemplo de código

```
int i
if true {
  float i = 0.0
}
for i = 0, i < 2, i = i + 2 {
  int a
}
bool a = true
```

Saída esperada

```
int var: i
if: true
then:
  float var: i = 0.0
for: = i 0, < i 2, = i + i 2
do:
  int var: a
bool var: a = true
```

Tratamento de erros

8. Variáveis somente são visíveis dentro de seus escopos e seus escopos filhos, levando a erros semânticos como listados no item 3 no caso de tentativas indevidas de uso de variáveis.

Versão 0.7 – Funções

Esta versão adiciona à linguagem as seguintes características relacionadas a funções:

I. Funções podem ser declaradas para uso posterior no código.
II. Funções podem ser definidas, o que envolve a declaração da assinatura da função e de seu corpo.
III. Funções podem ser usadas em testes e atribuições.
IV. Funções podem ser declaradas e definidas nos mesmos lugares onde variáveis podem.
V. Funções têm zero ou mais parâmetros
VI. Ao fim do código, todas as funções declaradas precisam ter sido definidas.
VII. Todas funções terminam com uma chamada de retorno com a palavra reservada ret.
VIII. Os nomes de funções seguem as mesmas regras que os nomes de variáveis.
IX. Os corpos de funções possuem os próprios escopos.
X. Os parâmetros da função funcionam de forma similar a variáveis declaradas no corpo da função.
XI. Declarações de funções não geram mensagens de saída.
XII. Na saída padrão, o corpo de cada função altera a tabulação em dois espaços em branco cumulativos.

Exemplo de código

```
bool fun f ()
bool fun f () {
ret false
}
if f() {
  int a = 0
  int fun f2 ( int x ) {
          int a
         a = x + 1
         ret a
  }
  a = f2 ( a)
}
```

Saída esperada

```
int fun: f (params: )
  ret false
if: f[0 params]
then:
  int var: a = 0
  int fun: f2 (params: int x)
    int var: a
    = a + x 1
    ret a
  = a f2[1 params] a
```

Tratamento de erros

9. Uma função deve ter apenas um retorno, o qual deve ser feito na última linha de seu corpo. A ausência de um retorno ou a presença de outras linhas de código deve gerar um erro de sintaxe.
10. Uma função declarada mas nunca definida até o fim de seu escopo deve gerar uma mensagem de erro semântico. Exemplo:
```
fool fun myfun()
ctrl+d
[Line 2] semantic error: function myfun is declared but never
defined
```
11. No caso de redeclaração ou redefinição de uma função em seu mesmo escopo original, um erro semântico deve ser gerado no mesmo estilo do item 4. A mensagem pode chamar ambas como `re-definition of function`.
12. No caso de um parâmetro de tipo incompatível, um erro semântico deve ser emitido. Exemplo:
```
int a
int fun f (int x, int y)
a = f (0.0, 0)
[Line 3] semantic error: parameter x expected integer but received
float
```
13. No caso de parâmetros faltantes ou excesso de parâmetros em uma chamada de função, erros semânticos devem ser gerados. Exemplo:
```
bool b
bool fun x (int a__)
b = x()
[Line 3] semantic error: function x expects 1 parameters but
received 0
b = x(1, 2, 3)
[Line 3] semantic error: function x expects 1 parameters but
received 3
```
14. No caso de discordância sobre o número de parâmetros entre a declaração e a definição da função, um erro de redefinição deve ser gerado como no item 11. O mesmo deve ser feito para o caso de discordância entre os nomes do parâmetros.

Versão 0.8 – Arranjos

Esta versão da linguagem adiciona a possibilidade de declaração de arranjos de tipos básicos na linguagem.

Os índices de arranjos são sempre valores inteiros.

Exemplo de código

```
int a(10)
bool parity(10)
int i
for i = 0, i < 10, i = i + 1 {
      a(i) = i
      if a(i) / 2  == 0 {
            parity(i) = true
      } else {
            parity(i) = false
      }
}
```

Saída esperada

```
int array: a (size: 10)
bool array: parity (size: 10)
for: = i 0, < i 10, = i + i 1
do:
  = [index] a i i
 if: == / [index] a i 2 0
  then:
    = [index] parity i true
  else:
    = [index] parity i false
```

Tratamento de erros

15. No caso de um índice de um tipo outro que inteiro, um erro semântico deve ser gerado. Exemplo:
    ```
    int a(10)
    a(false) = 1
    [Line 2] semantic error: index operator expects integer but received boolean
    ```
16. No caso da declaração de um arranjo com um tamanho que não seja um valor inteiro, um erro de sintaxe deve ser gerado.

Versão 1.0 - Ponteiros

A versão 1.0 da linguagem Łukasiewicz inclui a declaração e uso de ponteiros sobre os tipos básicos, arranjos e ponteiros. Ponteiros são identificados pela palavra reservada ref. O lvalue de qualquer variável pode ser acessado pelo operador addr.

Referências não são feitas para arranjos nem funções, sendo usadas apenas para variáveis e itens de arranjos.

Ponteiros podem ser usados como parâmetros de funções e retorno de funções.

Exemplo de código

```
int a(10)
int i = 0
int ref mypointer
int ref pointers(2)
mypointer = addr i
i = ref mypointer + 1
pointers(0) = mypointer
pointers(1) = addr a(3)
int ref ref doublepointer
doublepointer = addr pointers(0)
```

Saída esperada

```
int array: a (size: 10)
int var: i = 0
int ref var: mypointer
int ref array: pointers (size: 2)
= mypointer [addr] i
= i + [ref] mypointer 1
= [index] pointers 0 mypointer
= [index] pointers 1 [addr] [index] a 3
int ref ref var: doublepointer
= doublepointer [addr] [index] pointers 0
```

Tratamento de erros

17. No caso de operações binárias com tipos diferentes, erros semânticos devem ser emitidos. Exemplo:
    ```
    int i = 0
    int ref point
    point = i
    [Line 3] semantic error: attribution operation expects integer
    pointer but received integer
    ```
18. O operador unário ref espera um valor que seja ponteiro de algo. Caso o valor não seja um ponteiro, um erro semântico deve ser disparado. Exemplo:
    ```
    int i
    i = ref 0
    [Line 2] semantic error: reference operation expects a pointer
    ```

```
i = ref i
[Line 3] semantic error: reference operation expects a pointer
```

19. O operador unário addr espera uma variável ou item de arranjo. Caso contrário, um erro semântico deve ser gerado. Exemplo:

```
bool ref b
b = addr true
[Line 2] semantic error: address operation expects a variable or
array item
```

Termos para saída do compilador e mensagens de erro

Versão 0.1

Saída padrão

nameT: [a-zA-Z][a-zA-Z0-9_]*
intT: [0-9]+
typeT: int
var *typeT*: *nameT* {= *intT*}?{, *nameT* {= *intT*}?}*

Erros

[Line N] lexical error: unknown symbol X
[Line N] syntax error
[Line N] semantic error: undeclared variable X
[Line N] semantic error: re-declaration of variable X

Versão 0.2

Saída padrão

typeT: int | float | bool
boolT: true | false

Erros

op: addition | subtraction | multiplication | division | attribution | unary minus | equal | different | greater than | less then | greater or equal than | less or equal than | and | or | negation
type: integer | float | boolean
[Line N] semantic error: *op* operation expected *type* but received *type*

Versão 0.3

Saída padrão

[*typeT*]

Versão 0.4

Saída padrão

if:
then {
}
} else

Erros

op: *op* | test

Versão 0.5

Saída padrão

for: , ,
do:

Versão 0.7

Saída padrão

typeT fun: *nameT* (params: {*typeT nameT*{, *typeT nameT*}*}?)
ret
nameT[N params]

Erros

[Line N] semantic error: function *nameT* is declared but never defined
[Line N] semantic error: parameter *nameT* expected *typeT* but received *typeT*
[Line N] semantic error: function *nameT* expects N parameters but received N

Versão 0.8

Saída padrão

typeT array: *nameT* (size: N)
[index]

Erros

[Line N] semantic error: index operator expects integer but received { boolean | float }

Versão 1.0

Saída padrão

typeT {ref}* var: *nameT*
[addr]
[ref]

Erros

[Line N] semantic error: attribution operation expects *typeT* {pointer}* but received *typeT* {pointer}*
[Line N] semantic error: reference operation expects a pointer
[Line N] semantic error: address operation expects a variable or array item