|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | |
|  | |  | |
| Relatório Final  *Compiladores – Licenciatura em Engenharia Informática* | | | |
|  |  | |  |
|  |  | | Gonçalo Almeida, 2020218868  João Santos, 2020218995 |

Índice

[Meta 2 - Gramática reescrita 2](#_Toc121608525)

[Meta 2 e 3 - Algoritmos e estruturas de dados da *AST* e da *Symbol Table* 2](#_Toc121608526)

[Meta 4 – Geração de código 2](#_Toc121608527)

# Meta 2 - Gramática reescrita

A gramática para ser reescrita passou por 4 passos:

* Passar do formato EBNF para BNF
  + Passámos o formato {A} para um novo não terminal B, com B = ε | AB (por exemplo, {COMMA ID}, da produção dos FieldDecl passou para FieldCommaId = ε | COMMA ID FieldCommaId)
  + Passámos o formato [A] para um novo não terminal B, com B = ε | A
  + Passámos o formato A = B(C|D|…)E para A = BCE | BDE | … (como por exemplo, ( MethodInvocation | Assignment | ParseArgs ) nos Statements)
* Escolhemos retirar as produções nulas (do formato A = BCD, C = ε | E passámos para A = BD | BED) para obter um maior controlo sobre certos casos como Statement = LBRACE { Statement } RBRACE para Statement = LBRACE RBRACE | LBRACE Statement RBRACE | LBRACE MultipleStatements Statement RBRACE, onde necessitávamos de saber o número de statements num *Block.*
* Para as expressões (Expr), fizemos um ramo extra, ExprNoAssign, onde não se podem fazer assigns; em concreto, em qualquer expressão que não seja, parêntesis, Assignments, MethodInvocation e ParseArgs, não pode haver assigns. Isto evita situações do tipo a + b = c, permitindo na mesma a + (b = c).
* Adicionar as prioridades e associatividades necessárias para remover a ambiguidade da gramática, com %left quando um token é associativo à esquerda e %right quando um token é associativo à direita, sendo a ordem pela qual aparecem crescente de prioridade. Para distinguir o menos e o mais unário dos não unários, adicionámos “%prec UNARY” aos unários, de forma a aplicar a sua prioridade e associatividade corretamente.

# Meta 2 e 3 - Algoritmos e estruturas de dados da *AST* e da *Symbol Table*

A nossa *AST* é construída com nós, seguindo a seguinte estrutura: um tipo, um valor, um tipo convertido e um inteiro (Para facilitar a impressão), os parâmetros (caso represente uma função), um ponteiro para o seu filho e outro para o seu irmão, dois inteiros correspondentes à linha e coluna para futura deteção de erros e um inteiro para impedir verificação duplicada de nós com o mesmo nome.

A nossa *symbol table* é baseada em 3 estruturas: a tabela de símbolos em si que possui um tipo, um nome, os parâmetros e os símbolos (caso seja uma função) e o ponteiro para a próxima tabela; Cada símbolo tem um nome, um tipo, os parâmetros (caso seja uma função) e o ponteiro para o próximo símbolo; e cada parâmetro, possui um nome, um tipo e o ponteiro para o próximo parâmetro.

A criação da AST baseia-se na criação da estrutura nó, especificada acima, para cada tipo, apesar de que cada um tem tipos diferentes de nós filhos. Dando o exemplo de nós dos tipos if e while, estes tipos possuem funções específicas (add\_if e add\_while) pois dependendo se é um statement com apenas if ou if\_else; já o ciclo while

No nosso código, inicialmente, declaramos duas tabelas de símbolos, uma global e uma tabela que representará a primeira função do programa. Na tabela de símbolos global estão presentes todas as variáveis globais e todas as funções. Na tabela de símbolos de cada função estão, por ordem, o retorno, os seus parâmetros, e todas as variáveis, usadas e declaradas, e funções, usadas na mesma.

Na análise descendente adicionamos os símbolos às tabelas de símbolos e na análise ascendentes anotamos e fazemos a deteção de erros.

# Meta 4 – Geração de código

A geração do código passa por 2 fases:

1. Declaração de constantes, strings, variáveis globais e três funções externas ao programa;
2. Declaração de todas as funções presentes no programa, começando com a denominada de “main”.

Na primeira fase definimos constantes que nos permitem imprimir valores booleanos, inteiros, valores decimais e strings; strings, que aparecem no programa, como constantes para impedir que haja repetição de declaração de variáveis, isto é, se uma string for usada várias vezes não acontece o caso de esta ser definida várias vezes mas sim é carregado o valor desta; variáveis globais, que vamos buscar à tabela de símbolos global, especificada no ponto anterior; e três funções externas aos programas, sendo estas o printf e o atoi do C e uma função que criámos que imprime booleanos.

Na segunda fase geramos o código llvm com base na AST, isto é, na mesma forma que na meta 3 verificamos cada nó e retornamos um tipo, nesta meta verificamos cada nó e geramos o código llvm correspondente ao que levou à criação do nó em primeiro lugar.

Para a realização da segunda fase criámos funções que se categorizam em 4 tipos:

1. Conversão de strings, inteiros e booleanos para valores admitidos pela linguagem;
2. Impressão de booleanos, inteiros, valores decimais, strings e funções;
3. Carregamento de valores passados como parâmetro, isto é, através de um counter criamos variáveis que guardam os valores presentes nos nós;
4. Comparação de booleanos , de inteiros e valores decimais e dos dois nós filhos do nó passado como parâmetro, sendo que todas estas funções usadas principalmente em expressões.