UNIVERSIDADE DE ÉVORA

Compiladores
2015/2016

Docente:

Pedro Patinho

Discentes:

Pedro Jacob - 27421

André Figueira - 31626

Introdução

Com este trabalho tivemos como objectivo criar um compilador para a linguagem Yalang. Isto sendo possível através da criação do analisador lexical, sintáctico, gerador de código MIPS, implementação da symbol table e APT e analisador semântico.

Com o conhecimento adquirido durante este semestre, iremos tentar aplicar da maneira que acharmos correcta e melhor o que nos for pedido.

Análise Lexical

A analise lexical é composta por todos os elementos que irão ser utilizados num programa.

Estes elementos são a especificação de palavras reservadas, expressões regulares, etc... Estas palavras representam algo que vai fazer parte da lógica do programa, cujo nome faz parte da linguagem em si.

Exemplo: if, else, then, ==, <=, >=, and, or, not.

Estes simbolos não podem ser usadas pelo utilizador como por exemplo um ID de uma declaração: **if : int = 2;**

Os literais que são tipos constantes: estes são NUM_LIT, NUM_FLOAT_LIT, STRING_LIT, BOOL_IT

Em relação aos \n e \t decidimos que seria melhor ignorar estes pois seria mais simples que criar uma produção própria no analisador sintáctico que detecta se esses símbolos.

ID [a-zA-Z][a-zA-Z0-9_]*
INT [0-9]*
INT_FLOAT [0-9]*\.[0-9]+([eE][\+\-]?[0-9]+)?
BOOL "true"|"false"
STRING \"[a-zA-Z0-9_].*\"

As expressões regulares também fazem parte desta analise. Estas representam o que é válido num tipo, por exemplo um INT não pode ser uma String mas uma String é reconhecida como String se e só se estiver entre “ ”.

```
{ID}            {yylval.id = strdup(yytext); return ID;                 }

{INT_FLOAT}     {yylval.valfloat = atof(yytext); return NUM_FLOAT_LIT;
    }

{INT}           {yylval.val = atof(yytext); return NUM_LIT;     }

{STRING}    {yylval.str  = strdup(yytext) ; return STRING_LIT;      }
```

Análise Sintáctica e Análise Semântica

A analise sintáctica notou-se um problema ao inicio visto que não compreendíamos bem como estruturar a gramática.

Após um estudo decidimos implementar a gramática da melhor maneira que nós pensamos, tendo em mente que no futuro possivelmente iriam surgir mudanças.

Decidimos começar todo o programa com o não terminal **input** que corresponde onde o programa irá começar a ser analisado.

Se for detectado algo que não corresponde a estrutura da gramática então irá gerar um erro de sintaxe como esperado. Se for válido então faz a representação correcta (os **printf** auxiliares permitem fazer a verificação correcta por parte do utilizador) no output do terminal.

Enquanto se percorre a gramática vão se gerando os nós da APT e fazendo a análise semântica juntamente com a tabela de símbolos e APT, em que vai adicionando para a symbol table os símbolos.

Nota: O byson recebe nos includes para além das bibliotecas comuns, a interface da apt **“apt.h”** em que em cada produção pode gerar um nó, por exemplo **new_stmts_cond($1)** que recebe o não terminal em $1.

Reparou-se também a constante mudança da gramática seja para correcção de alguns erros ou de funcionalidades que foram esquecidas.

Existe para cada produção uma chamada de uma função para gerar nós da APT para todos os não terminais pois é o que nos fazia mais sentido.

Os não terminais escolhidos foram:

- condições
- expressões
- operadores
- expressões booleanas
- tipos
- múltiplos ids
- afetações
- declarações
- ciclos
- statements
- chamadas de funções
- corpo de ciclos e corpo de funções

As razões por trás destas escolhas estão descritas no enunciado do trabalho.

Por exemplo existirem vários tipos de declarações, statements e expressões fez nos sentido que significaria várias produções numa gramática, logo considerados **não terminais**. Nota-se também o caso em que, por exemplo, declarações pode gerar, ou não, novas declarações. Portanto fez mais sentido serem escolhidos como **não terminais,** mesmo se aplica para o corpo e funções ou de ciclos.

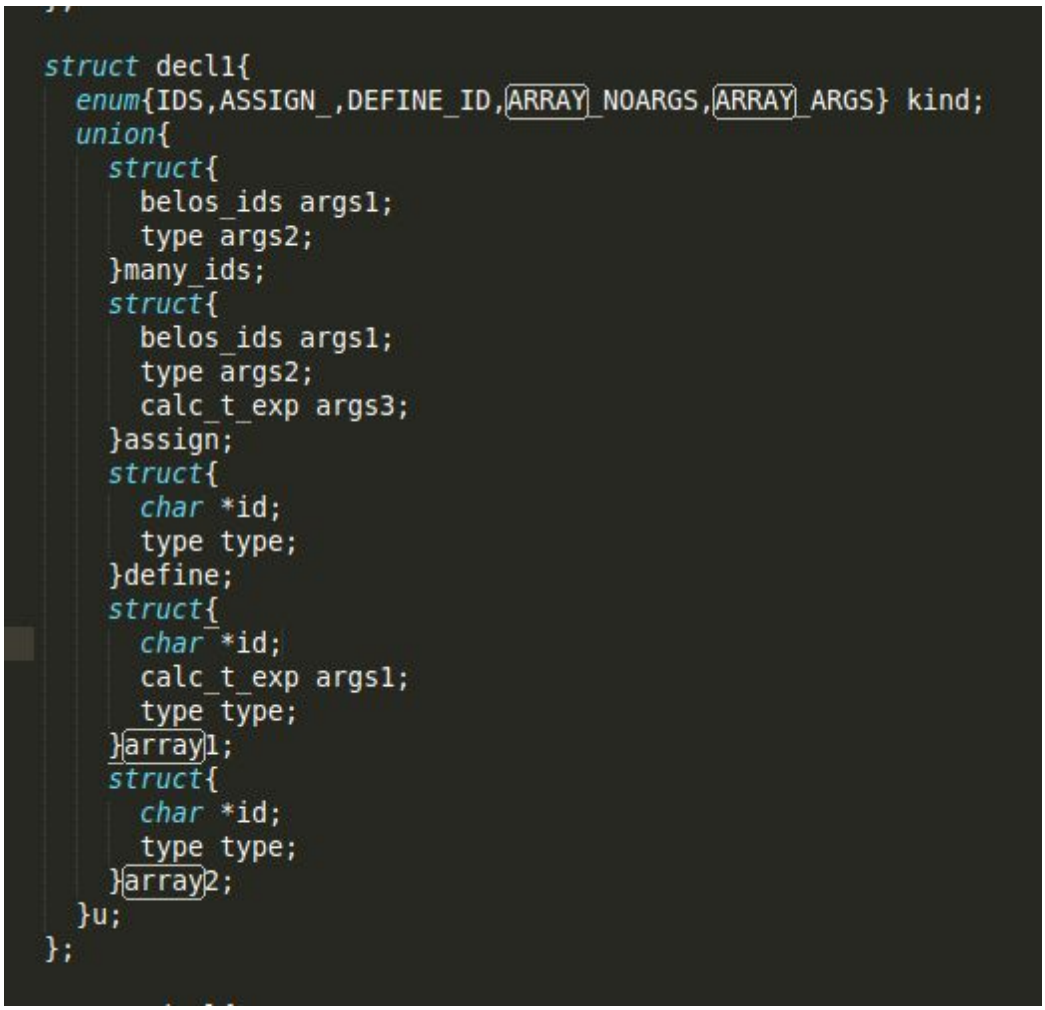
Nota: Expressões foi o mais simples de implementar pois nos baseamos na calculadora que implementamos nas aulas práticas, mesmo se aplica para a implementação da APT.

APT

Composta por dois ficheiros: **apt.h** e **apt.c**

A apt.h corresponde às assinaturas das funções e todas as structs a serem usadas. Estas estruturas correspondem a todos os não terminais na análise sintáctica.

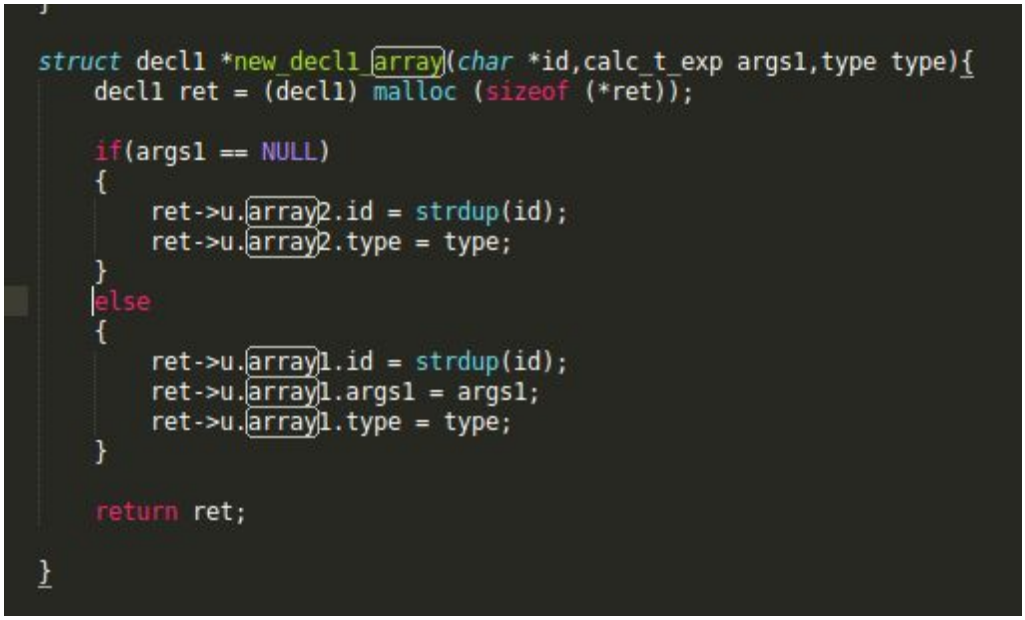
A estrutura comum escolhida em todos é semelhante à usada na calculadora que implementamos:



Composto por enumeração que espefica o tipo que nó, e composto por várias structs que representa o nó. Exemplo: array1 representa um array do genero a : int[ARGS1];

Todas as outras structs tem uma representação semelhante há excepção de 3. O ciclo, tipo e operador, que só contém uma enumeração.

O apt.c é nada mais que a implementação de todas as funções descritas no fim de **apt.h** em que recebe nos argumentos o tipo e cria uma nova struct em que inicializa os campos da struct correspondente. **Exemplo:**



Symbol Table

Decidimos implementar a symbol table sobre uma hashtable como sugerido nas aulas. Esta tabela está contida no próprio YA.y,

Juntamente com este relatório é enviado também uma versão antiga da symbol table com uma estruturação mais complexa mas devido a falta de tempo não conseguimos implementar portanto optamos por fazer uma versão no próprio YA.y. Mas embora esteja com alguns erros, decidimos enviar também juntamente com esta nova menos complexa.

A symbol table quando percorre a APT e repara num ID, o seu TIPO e ID são adicionados para a hashtable.

Nesta symbol table simples não foram implementados scopes mas o funcionamento é o seguinte:

- Sempre que se encontra uma função o seu nome e tipo são adicionados para a symbol table e em seguida gera-se um novo scope (um novo contexto) ou seja uma nova hashtable (aponta para outra hashtable e esta tem um apontador para a original, para saber para onde voltar) em que os argumentos da função são adicionados e tudo o que se segue, ou seja, dentro do corpo, é adicionado para a symbol table.

- Quando a função chega ao fim tudo é removido desta nova hashtable, e volta para a hashtable original (usando um apontador que aponta para a antiga hashtable) , voltando ao procedimento normal.

A symbol table permite também que faça análise de tipos, isto é, quando se faz look_up de um simbolo sabe-se que pode gerar erro se os tipos não corresponderem.

Nota: Quando repara numa declaração ou numa produção com o tipo **ids ou funcdeclar,** por exemplo:

ids DOUBLEPOINT type END_OF_STMT

é necessário tratamento especial pois **ids** contém vários ID's portanto a função tem de percorrer esse apontador e extrair a informação necessário para colocar na hashtable.

```
// declaracoes do genero, a,b,c,d,e : int;
void table_insert_2( belos_ids ids,type type)
{
    belos_ids root = ids;

    while(root != NULL)
    {
        table_insert_1(root->u.many_ids.id,type);
        root = root->u.many_ids.more_ids;
    }
}

void insertFunc(funcdeclar funcdeclar, type type){
    table_insert_1(funcdeclar->u.id,type);

}
```

Conclusão

Embora não tenhamos concluído a totalidade dos objectivos que foram definidos, conseguimos resolver parcialmente o que foi proposto. Como tal o nosso conhecimento sobre a estruturação de um compilador certamente aumentou.

Os objectivos não foram concluídos na sua totalidade, devido a outras avaliações e por um deste grupo ter um horário de trabalho complicado portanto foi algo que dificultou a realização do trabalho.